Acta Informatica (2023) 60:209–255
https://doi.org/10.1007/s00236-023-00438-4

ORIGINAL ARTICLE

Toward a theory of program repair

Besma Khaireddine[1] · Aleksandr Zakharchenko[2] · Matias Martinez[3] · Ali Mili[2]

Received: 16 December 2020 / Accepted: 19 February 2023 / Published online: 27 March 2023

Abstract
To repair a program does not mean to make it (absolutely) correct; it only means to make it more-correct than it was originally. This is not a mundane academic distinction: given that programs typically have about a dozen faults per KLOC, it is important for program repair methods and tools to be designed in such a way that they map an incorrect program into a more-correct, albeit still potentially incorrect, program. Yet in the absence of a concept of relative correctness, many program repair methods and tools resort to approximations of absolute correctness; since these methods and tools are often validated against programs with a single fault, making them absolutely correct is indistinguishable from making them more-correct; this has contributed to conceal/obscure the absence of (and the need for) relative correctness. In this paper, we propose a theory of program repair based on a concept of relative correctness. We aspire to encourage researchers in program repair to explicitly specify what concept of relative correctness their method or tool is based upon; and to validate their method or tool by proving that it does enhance relative correctness, as defined.

1 Introduction

As a field of research and development, the discipline of program repair has achieved great strides over the past decades, producing a continuous stream of increasingly sophisticated engineering solutions spanning several programming languages and several categories of faults [10, 11, 18, 19, 23, 28, 34, 36, 39–43, 45, 51, 53, 57, 60, 63–71, 74, 74, 76, 77, 79].

This work is partially supported by NSF under grant number DGE 1565478.

✉ Ali Mili
mili@njit.edu

Besma Khaireddine
khaireddine.besma@gmail.com

Aleksandr Zakharchenko
az68@njit.edu

Matias Martinez
matias.sebastian.martinez@gmail.com

[1] University of Tunis El Manar, Tunis, Tunisia

[2] NJIT, Newark, NJ, USA

[3] University of Valenciennes, Valenciennes, France

In [19], Gazzola et. al. present a sweeping survey of program repair, spanning more than two decades and encompassing more than 100 papers, and conclude that "it is important to improve the maturity of the field and obtain a better understanding of useful strategies and heuristics".

To repair a program does not mean to make it (absolutely) correct; it only means to make it more-correct (in some sense) than it was originally. This is not a mundane academic distinction: given that typical software products have thousands of KLOC, and about a dozen faults per KLOC, it is rather critical that program repair methods and tools be designed in such a way that they transform an incorrect program into a more-correct, while still possibly incorrect, program. Hence, the validation test in program repair ought to be based on relative correctness (the property of a program to be more-correct than another with respect to a specification) rather than absolute correctness. Yet, in the absence of a concept of relative correctness, program repair methods and tools have resorted to various approximations of absolute correctness; some of the criteria used to select candidate repairs may sound/look like relative correctness, but we reserve the term *relative correctness* to ordering relations between programs that satisfy specific litmus conditions, which we discuss in Sect. 4.1. The failure to distinguish between absolute correctness and relative correctness is concealed by the fact that many (though not all: [18, 57, 68, 69]) program repair methods and tools are tested against programs seeded with a single fault at a time; so that making them absolutely correct is indistinguishable from making them more-correct.

In this paper, we propose a theory of program repair based on the following premises:

- To repair a program P with respect to a specification R means to transform it into a program P' that is strictly more-correct than P with respect to R (in a sense to be defined).
- A theory of relative correctness ought to play for program repair the same role that theories of absolute correctness [16, 22, 25] play for program derivation: In the same way that the derivation of a program P from a specification R is judged by whether P is absolutely correct with respect to R, the transformation of a program P onto a program P' in the context of program repair ought to be judged by whether P' is strictly more-correct than P with respect to R.
- Any program repair method or tool ought to be based on an explicit definition of relative correctness, and ought to be validated by proving that it does enhance (relative) correctness as defined.

In Sect. 2, we present a brief/cursory survey of program repair, then we discuss why we feel that the absence of theoretical foundations for this discipline yields gaps/loopholes in its efficiency and effectiveness. In Sect. 3, we present simple mathematical foundations for program semantics, which we use in Sect. 4 to introduce a theory of program repair, based on a definition of relative correctness. In Sect. 5, we present a generic algorithm for program repair whose main thrust is to achieve absolute correctness by stepwise increments of relative correctness; we prove the correctness of this algorithm in Sect. 6 using Hoare's logic [25] and we illustrate its performance in Sect. 7 using common program repair benchmarks [30]. We conclude in Sect. 8 by summarizing our results, commenting on their potential to enhance the practice of program repair, discussing their shortcomings/ threats to their validity, then outlining some venues for further research.

2 State of the art: why do we need a theory?

In this section, we briefly survey some of the most representative program repair methods and tools, then we discuss in what way and to what extent a theory of program repair may enhance the effectiveness and efficiency of this discipline. For a detailed survey of this field, see [19]; following Gazzola et al. we distinguish between two broad families of program repair methods, which we review in turn below: *Generate-and-Validate* methods, and *Semantics-Based* methods.

2.1 Generate-and-validate methods

Generate-and-Validate methods of program repair proceed in two phases: in the first phase several modified versions of the program are generated according to various criteria/assumptions; in the second phase, these versions are tested against prespecified test suites to ensure that the correct behavior of the original program is preserved and incorrect behavior is corrected.

In [43, 73], Weimer at al. present *GenProg*, a program repair tool that generates candidate repairs using genetic programming. This tool uses three artifacts, namely the faulty program, a set of (positive and negative) test cases and (in some versions) a fitness function that is used to rank repair candidates. GenProg manipulates programs at different levels of abstraction, including the abstract syntax tree and the source code, and it uses quantitative/semantic criteria as well as qualitative/ syntactic criteria to select repair candidates; it favors repairs that involve minimal patches.

In [35], Kim et al. argue that because of its random mutation operators, GenProg is prone to generate too many non-viable patches, which impede its performance and its precision. Also, they propose a catalog of common faults, along with corresponding repair templates, which they have compiled from an analysis of more than 60 000 human-generated patches. Kim et al. implement their method in an automated tool, called *PAR*, for Pattern-based Automatic Program Repair.

In [48, 50], Long and Rinard present a program repair method (called *SPR*: Staged Program Repair), whose patch generation method relies on three techniques to reduce the search space: the first deploys a set of fault-specific *parameterized transformation schemas*. The second derives parameter values of the transformation schemas to ensure successful repair. The third technique aims to fine-tune the parameters of the transformation schemas. In [47], Long and Rinard integrate the staged program repair (SPR) method into a tool they call *Prophet*, which learns patch generation by analyzing a database of past successful patches.

In [39], Le et al. argue that semantic-based repair techniques suffer from weak specifications, and that repair methods are prone to overfitting because their patches are too specific to the test data. To enhance the generality of patches, they propose a three-pronged approach: a domain-specific language; an efficient search strategy; and favoring functions that are most likely to generalize. They validate their approach on standard benchmarks and on real-world programs.

In [76], Xin and Reiss introduce a program repair technique (*ssFix*) that retrieves patches from a code database, which includes segments from the program under repair as well as an auxiliary code repository. Once a suspicious statement is identified, *ssFix* performs syntactic code search to find candidate code segments that are structurally similar and conceptually related to the target statement and its local context (refered to as a *chunk*). The tool selects a

patch by matching repair candidates against the relevant chunk, then selecting the candidate that passes the validation test while minimizing discrepancies with the target chunk.

In [74], Wen et al. use empirical data from past research to fine-tune a program repair method that is based on two premises: first, that patch generation is better carried out at a fine level of granularity; second, that context information can be used to steer the mutation operators to patches that are most likely to be correct. They test their prototype, called *CapGen*, on the Defects4J benchmark.

In [70], Saha et al. introduce a program repair tool, called *Hercules*, whose distinguishing characteristic is its ability to repair faults that span multiple sites in the code, albeit under the special condition that the sites represent similar contexts and are repaired with similar code patches.

In [68], Rothenberg and Grumberg introduce a mutation-based program repair algorithm that is proved to be sound and complete, in the sense that it is guaranteed to return repairs that are minimal (in the number of mutations) and bounded-correct (i.e., correct within user-specified bounds in the number of iterations and the depth of recursion). The search space is reduced by limiting the search to unsatisfiable sets of constraints, using SAT and SMT solvers.

2.2 Semantics-based methods

The main difficulty with Generate-and-Validate methods is their predisposition to combinatorial explosion: they tend to generate too many candidate patches, without a commensurate gain in recall; it may be advantageous to push the validation step upstream, into the generation phase; this is the focus of semantics-based methods of program repair.

In [63], Nguyen et al. introduce *SemFix*, a method that proceeds in three steps: first it uses fault localization techniques to identify suspicious program locations. Then, it considers these faults by order of decreasing probability, and infers a local specification by deriving constraints on its behavior from controlled symbolic execution. Then, it deploys program synthesis techniques to derive a substitute for the faulty statement. Because SemFix relies on symbolic execution, it can only be applied to small programs that have no loops, a clear obstacle to scalability.

To overcome this shortcoming, Mechtaev et al. [57] introduce *Angelix*, a tool that relies on fault localization and program synthesis but uses a new artifact to represent the semantic constraints on the search space of patches: a set of execution traces of the program in which expressions are associated with constraints that ensure the successful execution of the program. They claim that this artifact is independent of the size of the program, hence it supports scalability.

In [39], Le et al. introduce *JFix*, a semantics-based program repair method that extends *Angelix* to deal with Java programs, using Symbolic PathFinder, a symbolic execution engine for Java programs. Though it extends Angelix, it is designed to support other repair methods as well.

In [12, 78], DeMarco et al. present a tool, called *Nopol*, that focuses on repairing conditions in if-statements and unguarded statements. This tool proceeds by instrumenting the program, executing it on the test data and keeping track of the values that the program state takes at different locations. This trace information is compiled into a SMT problem; if a solution exists, it is translated into a patch in the form of a modified if-condition or a newly generated guard.

In [23], Gupta et al. aim to fix *common C language errors*, which are syntactic faults that compilers are supposed to detect. But while a compiler merely declares that there is a fault, *DeepFix* attempts to fix it. Gupta et al. model the problem as a mapping from an input sequence (the erroneous program) to an output sequence (the fixed program), and design a neural net to solve it. A C compiler is used for patch validation.

In [72], Tan and Roychodhoury propose *ReliFix*, an automated tool for repairing software regressions that may result from program changes. They propose five criteria to guide the repair operation, which are: limit repairs to small changes; produce readable code; pass progression tests; pass previously failed regression tests; and ensure that no new regression is introduced.

In [31], Ke et al. propose a method, *SearchRepair*, which relies on software reuse technology. This method is based on a repository of human-generated code fragments that are mined from open source software. When a program fragment is suspected of being faulty, a specification of the desired behavior is derived as SMT constraints on local input behavior. Then, a constraint solver is invoked to find a match for the search key in the repository.

In [18], Frenkel et al. present an integrated method which combines attempts to prove that a program satisfies some properties with attempts to repair the program in case the proof fails. This method, called the *Assume-Guarantee-Repair* framework, is implemented on the basis of *assume-guarantee* reasoning, and applies to C-like programs extended with synchronous communication primitives. The *Assume-Guarantee* approach to the verification of composite systems proves the correctness of a composite system $M1||M2$ with respect to some property P by proving the correctness of $M1$ with respect to P under some assumption A, then proving that $M2$ satisfies A. The *Assume-Guarantee-Repair* approach of Frenkel et al. applies the *Assume-Guarantee* rule, and while seeking a suitable assumption A (through a learning-based method), deploys an incremental repair algorithm that seeks to repair the program if the verification attempt fails; Frenkel et al. characterize the effect of the repair algorithm as *bringing the system closer to satisfying the specification*, which sounds very similar to the idea of relative correctness put forth in this paper.

2.3 A focus on faults

Some recent approaches to program repair do not fall neatly into the characterization of Gazzola et al. [19] but have in common their focus on a formal analysis of faults and fault repair [6, 17, 29, 69, 75]. In [69], Rothenberg and Grumberg introduce the concept of *Must Location Set*, which is a set of program locations that includes at least one program location from each repair for an observed failure. A fault localization technique is said to be a *Must Algorithm* if it returns a must location set for each observed program failure. Rothenberg and Grumberg develop a fault localization algorithm and use it in a program repair algorithm to help reduce the search space without loss of recall. The concept of *must location* is reminiscent of the concept of *definite fault* introduced by Mili et al. [58]: a definite fault in an incorrect program is a program part that must necessarily be modified if the program is to be corrected.

In [51], Lou et al. critique the separation between two lines of research, namely fault localization and fault repair, and the fact that traditionally fault localization has been viewed as a means to achieve fault repair ends. They argue for a *unified debugging* approach, where fault repair is used to refine fault localization. They implement their approach in a tool, called ProFL, and highlight its performance on test benchmarks and on real software products.

In [9], Christakis et al. present a static technique that analyzes an error trace in a program and identifies a small set of statements within the trace that may be modified to satisfy

correctness conditions. Suspicious statements are ordered according to their likelihood of being the source of the observed failure.

2.4 Why do we need a theory?

Given that the discipline of program repair has been successful in producing a continuous stream of increasingly sophisticated methods and tools, it is legitimate to ask the question: Why do we need a theory?

Whereas they vary widely by how they perform patch generation, program repair methods differ fairly little in terms of how they perform patch validation. Most of them use a combination of the following techniques (see Table 1 for a brief/partial/cursory representative survey): search space pruning; test set reduction; fitness function ranking; testing for absolute correctness; and regression testing. We argue that all these techniques are flawed, as they are prone to loss of precision and loss of recall.

- *Search Space Pruning.* In the face of massive search spaces, some program repair methods renounce to analyze all the candidates delivered by the patch generation step, and prune out large swaths of the search space; unless extreme caution is applied in ensuring that excluded candidates are non-viable repairs, search space pruning carries the risk of **loss of recall**.
- *Test Set Reduction.* In the face of massive search spaces, program repair methods have an incentive to shorten the test of each candidate, so as to inspect the largest possible set of candidates. Short test suites increase the likelihood that a candidate patch passes the test while being incorrect, hence leading to **loss of precision**.
- *Absolute Correctness Test.* Absolute correctness is a sufficient condition of relative correctness, but not a necessary condition. Hence, testing candidate repairs for absolute correctness rather than relative correctness causes a **loss of recall**.
- *Fitness Function Ranking.* Program repair methods use a wide range of fitness functions, typically a combination of the number of passing tests and failing tests. Regardless of how they are defined, fitness functions define a total ordering to reflect what is essentially a (vastly) partial ordering: fitness functions cannot possibly provide sufficient conditions of relative correctness because any two programs can be ranked by fitness functions, even when they are not ranked by relative correctness. Consequently, regardless of how they are defined, fitness functions are prone to cause a **loss of precision**.
- *Regression Testing* Regression testing consists in ensuring that repair candidates preserve the correct behavior of the program under repair; because correct behavior is not unique, a program may preserve correctness without preserving correct behavior (see Fig. 3). Hence, regression testing defines a sufficient but unnecessary condition of relative correctness; as such, it is prone to cause a **loss of recall**.

While existing methods and tools suffer from inadequate precision and recall of their patch validation, we prove in Sect. 6 by means of Hoare logic that the generic algorithm presented in Sect. 5 has perfect precision and perfect recall. The theory that we present in this paper gives us the ontology that we need to specify what it means to have perfect precision and perfect recall, as well as the reasoning framework that enables us to prove precision and recall properties.

Note that while we advocate for a correctness-based approach to program repair (more specifically, an approach based on relative correctness), we are not offering a correctness verification method, as we do not know of a general method to prove that a program P' is more-correct than a program P with respect to a specification R (the relative correctness

Table 1 Patch validation: a Bottleneck of program repair

Method/tool	Patch generation	Patch validation techniques		
		Space pruning	Fitness function	Regression test
GenProg [41]	Mutation crossovers	Tournament selection	Weighted sum of pass/fail tests	List of patches that meet regression test
PAR [35]	Using patch templates	Tournament selection	Number of passing tests	First patch that meets regression test
SPR [49]	Transformation schemas	Schema-based pruning	Heuristics-based ranking	Regression-successful patches in time limit
Prophet [47]	Instantiates patch features		Derived from feature vector	Ranked list of patches that meet regr. test
CapGen [74]	Prob.-driven mutant generator	Context-aware prioritization	Mutation frequency, context similarity	Regression-successful patches in time limit
Cardumen [55]	Code template mining	Probabilistic space navigation	Number of failing tests	Test suite adequacy
SSFix [76]	Code chunks from database		Uses modification types and sizes	First test adequate patch
FixMiner [37]	Context matching	Iterative clustering		First test adequate patch
LSRepair [46]	Mutation-based		Heuristic prioritization	First test adequate patch
Elixir [70]	Schema-driven repair expressions	ML-based space pruning	Distance between repair expression, location	First test adequate patch
DLFix [42]	Context-based transformation with deep learning	Program analysis filtering	CNN classification of candidate patches	Regression test as selection criterion

equivalent of methods such as [25, 52, 59]). In [20], Ghardallou et al. present a method for program repair whose steps are all carried out by static analysis: verification that there is a fault; localization of the fault; repair of the fault; verification that the resulting program is more-correct than the original. Whereas all these steps are carried out by static analysis, the scope of the method is very narrow, hence it cannot be used as a general method for proving relative correctness.

3 Mathematics of relative correctness

3.1 Relational mathematics

We assume the reader familiar with elementary relational algebra [8], hence this section is not a tutorial on relations as much as it is an introduction of some terminology and notations. We represent sets in a program-like notation by writing variable names and associated data types; if we write S as: `X x; Y y;` then we mean to let S be the cartesian product $S = X \times Y$; elements of S are usually denoted by s and the X- (resp. Y-) component of s is denoted by $x(s)$ (resp. $y(s)$). When no ambiguity arises, we may write x for $x(s)$, and x' for $x(s')$, etc. A *relation* R on set S is a subset of $S \times S$. Special relations on S include the *universal relation* $L = S \times S$, the identity relation $I = \{(s, s) | s \in S\}$ and the empty relation $\phi = \{\}$. Operations on relations include the set theoretic operations of union, intersection, difference and complement; they also include the *domain* of a relation defined by $dom(R) = \{s | \exists s' : (s, s') \in R\}$, and the product of two relations R and R' defined by: $R \circ R' = \{(s, s') | \exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}$; when no ambiguity arises, we may write RR' for $R \circ R'$. The *pre-restriction* of relation R to set T is the relation denoted by ${}_{T\backslash}R$ and defined by: ${}_{T\backslash}R = \{(s, s') | s \in T \wedge (s, s') \in R\}$. The *converse* of relation R is the relation denoted by $\widehat{R}$ and defined by $\widehat{R} = \{(s, s') | (s', s) \in R\}$.

A relation R is said to be reflexive if and only if $I \subseteq R$, symmetric if and only if $R = \widehat{R}$, antisymmetric if and only if $R \cap \widehat{R} \subseteq I$, and transitive if and only if $RR \subseteq R$. A relation R is said to be *deterministic* (or: a function) if and only if $\widehat{R}R \subseteq I$, and *total* if and only if $RL = L$. A relation R is said to be a *vector* if and only if $RL = R$; vectors have the form $R = A \times S$ for some subset A of S; we use them as relational representations of sets. In particular, note that RL can be written as $dom(R) \times S$; we use it as a relational representation of the domain of R. We may sometimes, for the sake of convenience, use the same symbol to represent a set (say T) and the vector $(T \times S)$ that represents the same set, in relational form. Hence, for example, the restriction of relation R to set T can be written as $T \cap R$, where we interpret T as a vector. We admit without proof (a well-known property of functions) that if F and G are functions then $F = G$ if and only if $F \subseteq G$ and $GL \subseteq FL$.

3.2 Program semantics

Definition 1 Given two relations R and R', we say that R' *refines* R (abbrev: $R' \sqsupseteq R$ or $R \sqsubseteq R'$) if and only if $RL \cap R'L \cap (R \cup R') = R$.

Intuitively, this definition means that R' has a larger domain than R and that on the domain of R, R' assigns fewer images to each argument than does R. See Fig. 1. This relation is known to be a partial ordering, i.e., it is reflexive, transitive and antisymmetric [7]; also, if R and R' are deterministic then R' refines R if and only if R' is a superset of R. The refinement relation has lattice-like properties, which are discussed in [7].

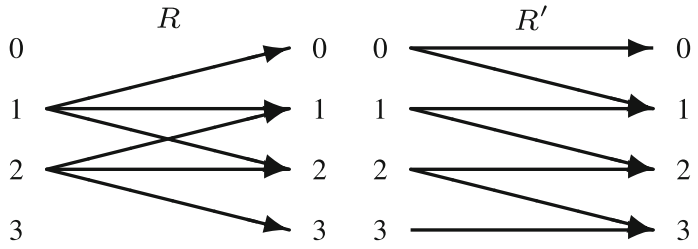


Fig. 1 $R' \sqsupseteq R$

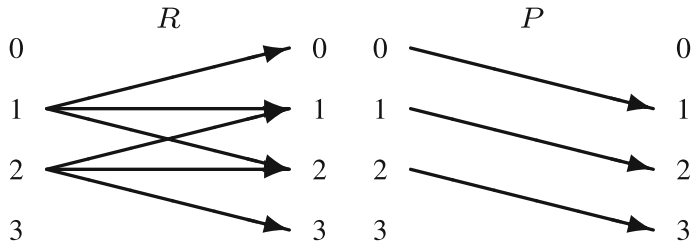


Fig. 2 Program P is correct with respect to R

Proposition 1 *Given two relations R and R', R' refines R if and only if for any relation Q, $R \sqsupseteq Q \Rightarrow R' \sqsupseteq Q$.*

Proof Necessity stems from the transitivity of the refinement relation [7]; sufficiency can be inferred by taking $Q = R$, which yields $R \sqsupseteq R \Rightarrow R' \sqsupseteq R$, which is equivalent to $R' \sqsupseteq R$ since $R \sqsupseteq R$ is a tautology (by reflexivity of the refinement relation). □

Given a program $\mathtt{p}$ on space S written in a C-like notation, we define the function of $\mathtt{p}$ (denoted by P) as the set of pairs (s, s') such that if program $\mathtt{p}$ starts execution in state s it terminates in state s'; it stems from this definition that $dom(P)$ is the set of states on which execution of P terminates. We may, when no ambiguity arises, refer to a program and its function by the same name, P.

Definition 2 A deterministic program $\mathtt{p}$ on space S is said to be *correct* with respect to specification R on S if and only if its function P refines R.

This definition is illustrated in Fig. 2: program P is correct with respect to R because for all the elements in the domain of R ($\{1, 2\}$), P is defined (terminates normally) and returns an output (2 for input 1, 3 for input 2) among those ($\{0, 1, 2\}$ for input 1, $\{1, 2, 3\}$ for input 2) that R mandates. The following Proposition, due to [59], helps set the context for Definition 3. We refer to the domain of $(R \cap P)$ as the *competence domain* of P with respect to R.

Proposition 2 *Due to* [59]. *Given a specification R and a deterministic program P, program P is correct with respect to R if and only if $(R \cap P)L = RL$.*

Although it looks very different, our definition of correctness is actually identical, modulo differences in the form of specifications, to traditional definitions of *total correctness* [22, 24, 52]. In [22, 24, 52], a program P is said to be *totally correct* with respect to a (pre/post) specification (ϕ, ψ) if and only if:

$$\forall s : \phi(s) \Rightarrow s \in dom(P) \wedge \psi(P(s)),$$

(since $s \in dom(P)$ means that program P terminates for initial state s). A relational specification R corresponds to the following (pre/post) specification:

- $\phi(s) \equiv s \in dom(R) \wedge s = s0$.
- $\psi(s) \equiv (s0, s) \in R$.

The following proposition provides that our definition of correctness is equivalent to traditional definitions of total correctness (modulo the difference in specification representation).

Proposition 3 *Given a specification R and a program P on space S, program P is correct with respect to R if and only if the following condition holds:*

$$\forall s : \phi(s) \Rightarrow s \in dom(P) \wedge \psi(P(s)),$$

where $\phi(s) \equiv s \in dom(R) \wedge s = s0$ and $\psi(s) \equiv (s0, s) \in R$ for some $s0$.

Proof *Proof of Sufficiency*. By replacing $\phi()$ and $\psi()$ by their expressions, we can simplify the condition of the proposition into:

$\forall s : s \in dom(R) \Rightarrow s \in dom(P) \wedge (s, P(s)) \in R$.

Since $(s, P(s))$ is by definition an element of P, this can be written as:

$\forall s : s \in dom(R) \Rightarrow s \in dom(P) \wedge (s, P(s)) \in (R \cap P)$.

By definition of domains, we can infer:

$\forall s : s \in dom(R) \Rightarrow s \in dom(P) \wedge s \in dom(R \cap P)$.

Since $dom(R \cap P) \subseteq dom(P)$, we infer:

$\forall s : s \in dom(R) \Rightarrow s \in dom(R \cap P)$.

By set theory, we infer: $RL \subseteq (R \cap P)L$; since the inverse inclusion is a tautology, we infer $(R \cap P)L = RL$.

Proof of Necessity. Since $(R \cap P)L \subseteq RL$ is a tautology, the condition $(R \cap P)L = RL$ is equivalent to $RL \subseteq (R \cap P)L$, which we interpret as follows:

$\forall s : s \in dom(R) \Rightarrow s \in dom(R \cap P)$

$\Rightarrow$ {Interpreting the definition of domain}

$\forall s : s \in dom(R) \Rightarrow \exists s' : (s, s') \in (R \cap P)$

$\Rightarrow$ {P is deterministic}

$\forall s : s \in dom(R) \Rightarrow \exists s' : s' = P(s) \wedge (s, s') \in R$

$\Rightarrow$ {substitution}

$\forall s : s \in dom(R) \Rightarrow \exists s' : s' = P(s) \wedge (s, P(s)) \in R$

$\Rightarrow$ {Interpreting the definition of domain}

$\forall s : s \in dom(R) \Rightarrow s \in dom(P) \wedge (s, P(s)) \in R$

$\Rightarrow$ {substituting $\phi()$ and $\psi()$}

$\forall s : \phi(s) \Rightarrow s \in dom(P) \wedge \psi(s) \in R$. □

4 A logic of relative correctness

4.1 Criteria for relative correctness

Before we propose a definition of relative correctness, we consider the question: *What constitutes a sound definition of relative correctness?* We have identified four properties we feel relative correctness ought to satisfy; for the sake of this discussion, we use the symbol $\sqsupseteq_R$ to represent the relation of relative correctness with respect to R.

– *Reflexivity and Transitivity, but not Antisymmetry.* We clearly want relative correctness to be reflexive and transitive; we do not want it to be antisymmetric because we want to admit that two programs be mutually more-correct while being distinct; in particular, we want to admit that two programs be both absolutely correct while being distinct (since correct behavior is not unique). Note that a more faithful name for this ordering is: *more-correct-than-or-as-correct-as*; for the sake of convenience we use the shorter version; whenever we want to exclude the clause *as-correct-as* we use the term *strictly more-correct*.
– *Culmination in Absolute Correctness.* We want relative correctness to culminate in absolute correctness, in the sense that if we keep making a program increasingly more-correct it will eventually become absolutely correct. In other words, an absolutely correct program with respect to some specification R ought to be more-correct than (or as correct as) any program with respect to the same specification. We write this property as:

$$P' \sqsupseteq R \Leftrightarrow (\forall P : P' \sqsupseteq_R P).$$

– *Relative Correctness and Reliability.* We define the reliability of a program in terms of two parameters, namely the specification with respect to which correct behavior is judged, and the probability distribution of the possible inputs (aka the program's *operational profile* [62]); for simplicity, we consider discrete probability distributions. Given a specification R, and a probability distribution $\theta()$ on the domain of R, the *reliability* of a program P with respect to R and $\theta()$, denoted by $\rho_R^{\theta}(P)$, is the probability that the execution of P on an element s of $dom(R)$ picked according to $\theta()$ succeeds, where successful execution of P on s with respect to R is defined as: $s \in dom(P) \wedge (s, P(s)) \in R$. We want to think that if P' is more-correct than P with respect to specification R then for any operational profile θ, P' is more reliable than P with respect to the same specification. But the opposite is not true, i.e., P' may be more reliable than P without being more-correct (P' is more reliable than P by virtue of satisfying the specification for inputs that occur more often): relative correctness is a logical property whereas reliability is a stochastic property; also we certainly do not want to think of *more-correct* as being just another name for *more reliable*. We write this as:

$$P' \sqsupseteq_R P \Rightarrow (\forall \theta() : \rho_R^{\theta()}(P') \geq \rho_R^{\theta()}(P)).$$

– *Relative Correctness and Refinement.* According to Proposition 1 and Definition 2, P' refines P if and only if for any specification R, if P is correct with respect to R, then so is P'. We want to use relative correctness to formulate a property between P' and P that expresses a relationship between P and P' even if P is not absolutely correct with respect to R: we want to define relative correctness in such a way that the following two statements are equivalent:

  – For any specification R, if P is correct with respect to R, then P' is correct with respect to R.
  – For any specification R, P' is more-correct than P with respect to R.

  Quantifying these properties over R, we write:

$$\forall R : (P \sqsupseteq R \Rightarrow P' \sqsupseteq R) \Leftrightarrow (P' \sqsupseteq_R P).$$

  Using Proposition 1, we can simplify this condition as:

$$P' \sqsupseteq P \Leftrightarrow (\forall R : P' \sqsupseteq_R P).$$

This formula is intuitively appealing, in the following sense: $(P' \sqsupseteq_R P)$ means that P' is more-correct than P for the specific purposes of specification R, whereas $P' \sqsupseteq P$ can be interpreted to mean that P' is more-correct than P regardless of what specification we consider.

We introduce a definition of relative correctness in the next section, and we prove in Sect. 4.3 that this definition meets all the conditions listed above.

4.2 Definitions and properties

The following definition, due to [58], applies to deterministic programs; it is a special case of definitions given in [13, 14] for non-deterministic programs.

Definition 3 Given a specification R and two deterministic programs P and P', we say that P' is *more-correct* (resp. *strictly more-correct*) than P with respect to R if and only if $(R \cap P')L \supseteq (R \cap P)L$ (resp. $(R \cap P')L \supset (R \cap P)L$).

We denote relative correctness (resp. strict relative correctness) with respect to R by $P' \sqsupseteq_R P$ (resp. $P' \sqsupset_R P$). To contrast relative correctness with correctness (Definition 2), we may refer to the latter as *absolute correctness*. The formula of relative correctness can be slightly simplified, as per the following proposition.

Proposition 4 *Program P' is more-correct than program P with respect to specification R if and only if $(R \cap P')L \supseteq (R \cap P)$.*

Proof Sufficiency can be proved by multiplying the two sides of the equation $(R \cap P')L \supseteq (R \cap P)$ by L on the right hand side and using the identity $LL = L$. Necessity stems from the identity $(R \cap P)L \supseteq (R \cap P)$. □

See Fig. 3. Specification R is shown in the middle. To the left, we show two programs, Q and Q', such that Q' is more-correct than Q with respect to R; to the right, we show two programs P and P' such that P' is more-correct than P with respect to R; the competence domain of each program is indicated by the ovals. Notice that Q' is more-correct than Q by virtue of imitating the correct behavior of Q, whereas P' is more-correct than P by virtue of a different correct behavior.

Proposition 5 *If Q' duplicates the correct behavior of Q with respect to R then Q' is more-correct than Q with respect to R.*

Proof The correct behavior of Q with respect to R is represented by $(R \cap Q)$. That Q' duplicates this behavior means $(R \cap Q) \subseteq Q'$. Since by definition $(R \cap Q)$ is also a subset of R, we infer: $(R \cap Q) \subseteq (R \cap Q')$, from which we infer: $(R \cap Q)L \subseteq (R \cap Q')L$. □

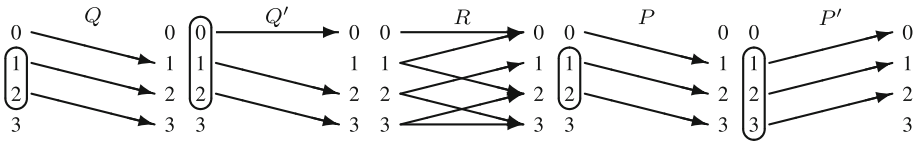


Fig. 3 $(Q' \sqsupseteq_R Q)$, $(P' \sqsupseteq_R P)$: preserving correctness $((R \cap P)L \subseteq (R \cap P')L)$ versus preserving correct behavior $((R \cap Q) \subseteq (R \cap Q'))$

Of course, the reverse implication does not hold: Fig. 3 shows a simple counter-example (programs P and P'). This proposition justifies our claim (Sect. 2) that using regression testing in patch validation leads to a loss of recall.

To illustrate this definition, we consider the following example, due to [15]: we let space S be defined by two nonnegative integer variables x and y, and we let R be the following specification on S:

$$R = \{(s, s') | x^2 \leq x'y' \leq 2x^2\}.$$

We consider the following candidate programs, denoted p_0 through p_7. Next to each program p_i, we represent its competence domain (CD_i). Figure 4 shows how these candidate programs are ranked by relative correctness with respect to R; this graph merely reflects the inclusion relationships between the competence domains.

p_0: `{x=1; y=-1;}`. $CD_0 = \emptyset$.
p_1: `{x=2*x; y=0;}`. $CD_1 = \{s | x = 0\}$.
p_2: `{x=x*x; y=0;}`. $CD_2 = \{s | x = 0\}$.
p_3: `{x=2*x; y=1;}`. $CD_3 = \{s | 0 \leq x \leq 2\}$.
p_4: `{x=2*x; y=2;}`. $CD_4 = \{s | x = 0 \vee 2 \leq x \leq 4\}$.
p_5: `{x=2*x; y=x/2;}`. $CD_5 = S$.
p_6: `{y=(x+1)/2; x=2*x;}`. $CD_6 = S$.
p_7: `{x=x*x; y=2;}`. $CD_7 = S$.

This example illustrates a number of properties:

- Note that this relation is not antisymmetric; so that two programs may be mutually related and still be distinct (such is the case for P_1 and P_2, for example).
- The top of the graph represents the programs that are (absolutely) correct with respect to specification R: P_5, P_6 and P_7.
- A program may be more-correct than another without imitating its correct behavior. For example, p_3 is more-correct than p_1 and yet it does not behave as p_1 on the competence domain of p_1.
- Whereas absolute correctness divides the set of candidate programs into two classes (correct, shown in green in Fig. 4 and incorrect, shown in red in Fig. 4), relative correct-

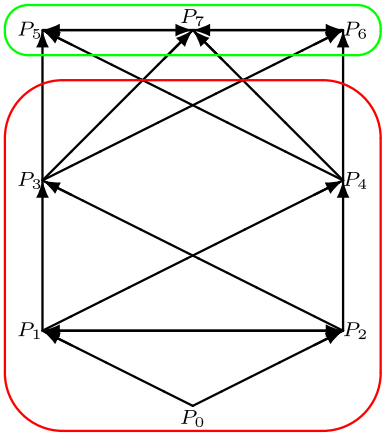


Fig. 4 Relative correctness relations

ness ranks candidate programs on a partial ordering, whose maximal elements are the absolutely correct programs.

4.3 Validation

In this section, we check that our definition of relative correctness meets the four conditions listed in Sect. 4.1.

- *Reflexivity and Transitivity, but no Antisymmetry.* Relative correctness is indeed reflexive and transitive (due to reflexivity and transitivity of set inclusion) but it is not antisymmertic, since $(R \cap P)L = (R \cap P')L$ does not necessarily imply $P = P'$.
- *Relative Correctness and Absolute Correctness.* Relative correctness culminates in absolute correctness, since an absolutely correct program P satisfies the condition $(R \cap P)L = RL$ (according to Proposition 2), hence its competence domain is maximal (hence a superset of the competence domain of any candidate program).
- *Relative Correctness and Reliability.* Relative correctness logically implies enhanced reliability; this is the subject of Proposition 6.
- *Relative Correctness and Refinement.* Propgram P' refines program P if and only if P' is more-correct than P with respect to any specification R; this is the subject of Proposition 7.

Proposition 6 *Given a specification R and two programs P and P', if program P' is more-correct than P with respect to R, then P' has a higher reliability than P with respect to R for any probability distribution $\theta()$ on $dom(R)$.*

Proof We consider a specification R and discrete probability distribution $\theta()$ on $dom(R)$; we let s be a random element of $dom(R)$ selected according to probability distribution $\theta()$. Execution of a program P on s is successful if and only if s is in the competence domain of P with respect to R. Hence, the reliability of P with respect to R and $\theta()$ can be written as:

$$\rho_R^{\theta()}(P) = \Sigma_{s \in dom(R \cap P)} \theta(s).$$

Clearly, larger competence domains yield greater values for $\Sigma_{s \in dom(R \cap P)} \theta(s)$, regardless of how $\theta()$ is defined. Hence:

$$P' \sqsupseteq_R P \Rightarrow (\forall \theta() : \rho_R^{\theta()}(P') \geq \rho_R^{\theta()}(P)).$$

□

Proposition 7 *Program P' refines program P if and only if P' is more-correct than P with respect to any specification R.*

$$(P' \sqsupseteq P) \Leftrightarrow (\forall R : P' \sqsupseteq_R P).$$

Proof *Proof of Necessity*. If $P' \sqsupseteq P$ then (because P and P' are both functions) $P' \supseteq P$, whence (by monotonicity of intersection and domain) $(R \cap P')L \supseteq (R \cap P)L$.

Proof of Sufficiency. From $(\forall R : (R \cap P')L \supseteq (R \cap P)L)$, we infer, by letting $R = P$, $(P \cap P')L \supseteq PL$. This, in conjunction with the set theoretic identity $(P \cap P' \subseteq P)$, yields (because $(P \cap P')$ and P are both functions), $P' \cap P = P$; from which we infer, by set theory $P' \supseteq P$; given that P' and P are both function, this yields $P' \sqsupseteq P$. □

4.4 Faults and fault repairs

Given that program repair is the art of diagnosing and removing faults, then surely it is worthwhile to ponder the question of what is a fault and what is a fault repair (i.e., when do we consider that we have removed a fault?). In [5, 38], Laprie et al. define a fault as the *adjudged or hypothesized cause of an error* [5]. Also, the IEEE Standard *IEEE Std 7-4.3.2-2003* [27] defines a software fault as *An incorrect step, process or data definition in a computer program*. In [21], Gopinath et al. define a fault as *an erroneous part of a program, the syntactic source of a semantically wrong behavior*. We feel that these definitions fall short of providing us a sound basis for reasoning about faults and fault repairs: the IEEE definition does not even try; the definition of Laprie et al. [5] depends on the definition of an *error*, which in turn depends on the availability of a specification of correct states at each step of a computation, clearly an unrealistic assumption; notwithstanding that it fails to define what is meant by *semantically wrong behavior*, the definition of Gopinath et al. assumes that each semantically wrong behavior has a unique syntactic source, clearly an unfounded assumption. In [6], Bergstra surveys some definitions of faults and compares them in terms of their relevance to the practice of debugging.

Following [15], we use the concept of relative correctness to define *faults* and *fault repairs*. The first observation we make in this regard is that any definition of fault must refer implicitly to a level of granularity at which we want to isolate faults. This typically varies in scale from the lexeme (variable name, operator, operand, etc.) to the expression, the elementary statement, or a subtree of the abstract syntax tree [19]. Following Gazzola et al., we adopt two definitions that determine the scale of our faults:

- A *syntactic atom* (or: *atom*) in program P is a fragment of source code of P at the selected level of granularity (lexeme, expression, statement, etc.).
- An *atomic change* in program P is a pair of source code fragments (a, a') such that a is a syntactic atom in P and a' is a code fragment that we can substitute for a without violating the syntactic correctness of P (i.e., the new program is syntactically correct).

We use the term *feature* in a program P to refer to one or more syntactic atoms in P; this concept is needed because some faults may involve more than one atom.

Definition 4 Due to [15]. Given a program P on space S and a specification R on S, a feature f of P is said to be a *fault* in P with respect to specification R if and only if there exists a substitute f' derived from f by atomic changes such that program P' obtained from P by replacing f with f' is strictly more-correct than P with respect to R. The pair (f, f') is said to be a *fault repair* of f in P with respect to R.

Note that according to this definition, if f_1 and f_2 are disjoint faults in P with respect to R, then (due to the transitivity of relative correctness) so is the aggregate $\langle f_1, f_2 \rangle$. This observation highlights the need to define a concept of *unitary fault*; this concept will, in turn, be used to quantify a program's degree of *faultiness*.

Definition 5 A fault f in program P with respect to specification R is said to be an *unitary fault* if and only if it includes a single atom, or it includes more than one atom but no subset of the atoms is a fault. The number of atoms in a unitary fault is called the *multiplicity* of the fault.

When we apply several atomic changes to a program to enhance its correctness, it is important to tell whether we are executing a single fault repair with higher multiplicity, or several fault repairs (with lower multiplicities). As an illustration, we consider the following space:

```
float x; float a[N+1];
```

We let *Sum* and P be, respectively, the following specification and program on space S:

$Sum = \{(s, s')|x' = \sum_{i=1}^{N} a[i]\}$.

$P =$ {int i = 0; x = 0; while (i < N) {x = x + a[i]; i = i + 1; }}

We consider the following feature $f = \langle 0, <\rangle$, wheren"0" is the initialization of variable i and "<" is the comparison operator in the loop's condition; the substitution $f' = \langle 1, \leq\rangle$ of f makes P strictly more-correct, hence f is a fault. To determine whether we are looking at two separate faults or a single two-atom fault, we consider the following programs:

$P_1' =$ {int i = 1; x = 0; while (i < N) {x = x + a[i]; i = i + 1; }}

$P_2' =$ {int i = 0; x = 0; while (i <= N) {x = x + a[i]; i = i + 1; }}

P_1' is obtained from P by the atomic change $(0, 1)$ in the initialization of i, and P_2' is obtained from P by the atomic change $(<, \leq)$ in the loop condition. In order to determine whether $f = \langle 0, <\rangle$ is a unitary fault, we must check whether 0 (in the initialization of i) alone is a fault, and whether $<$ (in the loop condition) alone is a fault. To this effect, we must check whether P_1' is strictly more-correct than P, and whether P_2' is strictly more-correct than P. The functions of P, P_1' and P_2' are:

$$P = \{(s, s')|a' = a \wedge x' = \sum_{k=0}^{N-1} a[k]\}.$$

$$P_1' = \{(s, s')|a' = a \wedge x' = \sum_{k=1}^{N-1} a[k]\}.$$

$$P_2' = \{(s, s')|a' = a \wedge x' = \sum_{k=0}^{N} a[k]\}.$$

The competence domains of P, P_1' and P_2' with respect to *Sum* are, respectively:

$CD = \{s|a[0] = a[N]\}$.

$CD_1' = \{s|a[N] = 0\}$.

$CD_2' = \{s|a[0] = 0\}$.

Since no inclusion relation holds between CD and CD_1', P_1' is not more-correct than P with respect to *Sum*; since no inclusion relation holds between CD and CD_2', P_2' is not more-correct than P with respect to R. If we let P' be the program derived from P by applying both atomic changes, then its is clear that P' is absolutely correct with respect to R, hence it is more-correct than P, P_1' and P_2'. See Fig. 5.

Now, we consider the following example: we let the space S of the program be defined by the following program variable:

```
int a[N+1];
```

and we let the specification *Reset* and program Q be defined as follows:

$Reset = \{(s, s')|a[0] = a'[0] \wedge \forall k : 1 \leq k \leq N : a'[k] = 0\}$.

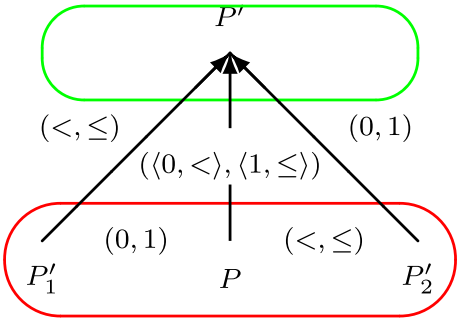


Fig. 5 A single unitary fault, of multiplicity 2

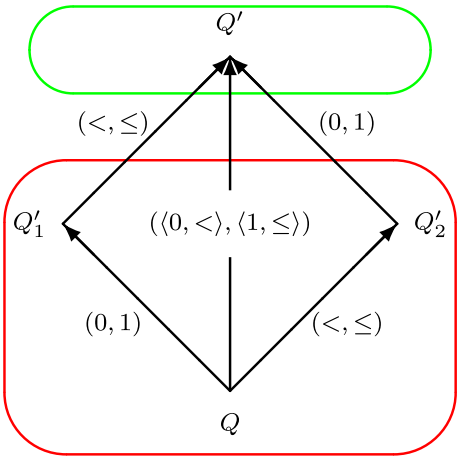


Fig. 6 Two unitary faults, of multiplicity 1

$Q =$ {int i = 0; while (i < N) {a[i] = 0; i = i + 1; }}

The function of Q is:

$$Q = \{(s, s') | a'[N] = a[N] \wedge \forall k : 0 \leq k \leq N - 1 : a'[k] = 0\}.$$

The competence domain of Q with respect to *Reset* is:

$$CD = \{s | a[0] = 0 \wedge a[N] = 0\}.$$

Because this is not the same as the domain of *Reset* (which is all of S), program Q is not correct with respect to R. We consider the following programs, obtained from Q by, respectively, changing the initialization of i to 1, changing the condition of the loop to $\leq$, and performing both changes.

$$Q'_1 = \{(s, s') | a[0] = a'[0] \wedge \forall k : 1 \leq k \leq N - 1 : a'[k] = 0 \wedge a[N] = a'[N]\}.$$
$$Q'_2 = \{(s, s') | \forall k : 0 \leq k \leq N - 1 : a'[k] = 0 \wedge a[N] = a'[N]\}.$$
$$Q' = \{(s, s') | a[0] = a'[0] \wedge \forall k : 1 \leq k \leq N : a'[k] = 0\}.$$

The competence domains of these programs with respect to *Reset* are given below:

$$CD'_1 = \{s | a[N] = 0\}.$$
$$CD'_2 = \{s | a[0] = 0\}.$$
$$CD' = S.$$

Considering the inclusion relations between CD, CD'_1, CD'_2 and CD', we infer that Q'_1 and Q'_2 are more-correct than Q, and that Q' is more-correct than all of Q, Q'_1 and Q'_2. This is illustrated in Fig. 6. The contrast between Figs. 5 and 6 illustrates the difference between a single unitary fault of multiplicity 2 and two unitary faults of multiplicity 1.

4.5 Fault density and fault depth

If a program P has N faults with respect to specification R and we repair one of them, we do not necessarily end up with $(N - 1)$ faults: Let f_1 and f_2 be two faults in program P with respect to specification R, let (f_1, f'_1) be a fault repair for f_1 and let P' be the program obtained from P following this repair; just because f_2 is a fault in P does not necessarily mean that f_2 is a fault in P'; in addition, a feature f_3 may well be a fault in P' whereas it is not a fault in P. This discussion leads us to consider two distinct metrics of faultiness in a program.

Definition 6 Given a program P and a specification R on space S, the *fault density* of P with respect to R is the number of unitary faults in P; the *fault depth* of P with respect to R is the minimal number of unitary fault repairs that separate P from a correct program.

Below are noteworthy remarks about these metrics:

- These two metrics are distinct; just because we have N unitary faults does not mean we need to perform N unitary fault repairs to obtain a correct program.
- These two metrics are subject to different rules. For example, if P' stems from P by a unitary fault repair then we have the equation:
  $$depth(P) \leq depth(P') + 1.$$
  Equality holds if P' is on a minimal path from P to a correct program. By contrast, fault density varies arbitrarily after a unitary fault repair.
- We find that fault depth is a more meaningful measure of faultiness than fault density (even though in the literature fault depth is not used and fault density is used routinely as a measure of faultiness).

For a detailed discussion of the contrast between fault density and fault depth, the reader is referred to [33]. In this paper, we content ourselves with an illustrative example:

We consider the following space S, program P, specification Sum, set of syntactic atoms SA, and set of atomic changes AC:

S: `float x; float a[N+1];`

$Sum = \{(s, s') | x' = \sum_{i=1}^{N} a[i]\}$.

$P =$ {`int i = 0; x = 0; while (i < N) {x = x + a[i]; i = i + 1;}`}

$SA = \{0, <, i\}$.

$AC = \{(0, 1), (<, \leq), (i, i + 1)\}$.

We leave it to the reader to check that this program has two unitary faults:

- $f_1 = \langle 0, < \rangle$ (where 0 is the value assigned to variable `i` and $<$ is the comparison operator that appears in the loop condition), a unitary fault of multiplicity 2, for which a possible repair is $f_1' = \langle 1, \leq \rangle$.
- $f_2 = \langle i \rangle$ (the index of the array `a[]` in the loop body), a unitary fault of multiplicity 1, for which a possible repair is $f_2' = \langle i + 1 \rangle$.

Figure 7 shows the relative correctness properties between P, P_1' obtained by repairing f_1', P_2' obtained by repairing f_2' and P'' obtained by applying both transformations. Notice that P has a fault density of 2, but a fault depth of 1; just because it has two faults does not mean it is two unitary fault repairs away from absolute correctness; actually if both fault repairs are applied, we end up with another program, P'', which has a fault density of 2 and a fault

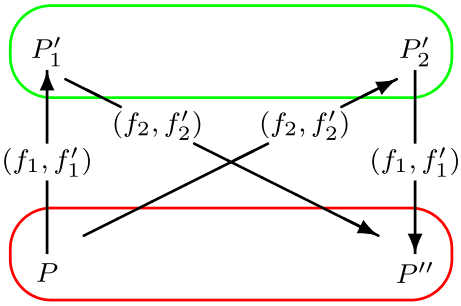


Fig. 7 Fault density (= 2) versus fault depth (= 1)

depth of 1. If anything, having two faults means that we have two opportunities to repair P; it does not mean that we have to apply two fault repairs.

To illustrate how density and depth vary as a function of granularity, atomic changes, and specifications, we consider the two programs P (sum of an array) and Q (initialize an array) discussed above. Also,

- For program P, we consider specification Sum.
- For program Q, we consider two specifications: specification $Reset$, presented above; and specification D defined as:
  $$D = \{(s, s') | a[0] \neq 0 \wedge a'[0] \neq 0 \wedge (\forall k : 1 \leq k \leq N : a'[k] = 0)\}.$$
- For atomic changes, we consider two sets:
  - $AC_0 = \{(0, 1), (<, \leq)\}$.
  - $AC_1 = \{(0, 1), (<, \leq), (i, i + 1)\}$.

Figure 8 shows the fault depth and fault density of programs P and Q for each combination of specification/atomic change set. The (program, specification) pairs are given in columns and the atomic change sets are given in rows. Note that fault density and fault depth are attributes of the (program/specification) pairs, whereas fault multiplicity is an attribute of individual faults. Hence, we represent the density and depth alongside each program/specification pair, and we represent multiplicity (function μ) alongside each individual fault.

The graphs drawn in the first and second column of Fig. 8 stem from the calculations of Sect. 4.4. The graphs drawn in the third column stem from the following analysis: First, we compute the competence domain of Q with respect to D.

$$Q = \{(s, s') | a'[N] = a[N] \wedge \forall k : 0 \leq k < N : a'[k] = 0\}.$$

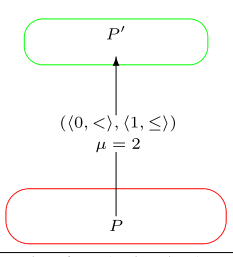
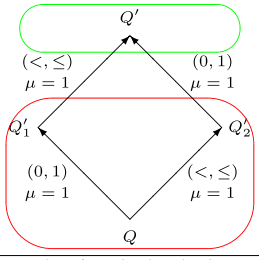
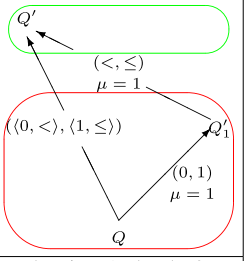
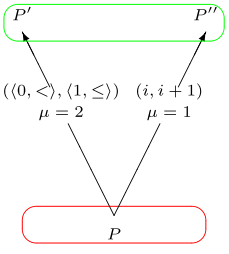
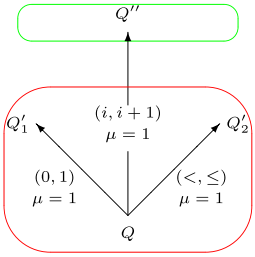
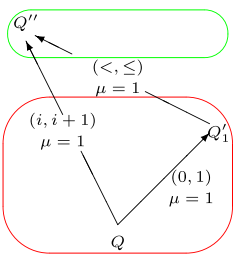
Program	P	Q	Q
Specification	Sum	$Reset$	D
AC_0= $\{(0, 1), (<, \leq)\}$			
Faultiness	density=1, depth=1	density=2, depth=2	density=1, depth=2
AC_1= $\{(0, 1), (<, \leq), (i, i + 1)\}$			
Faultiness	density=2, depth=1	density=3, depth=1	density=2, depth=1

Fig. 8 Density, depth and multiplicity

Hence,

$$Q \cap D = \{(s, s') | a[0] \neq 0 \wedge a'[0] \neq 0 \wedge \forall k : 0 \leq k \leq N : a'[k] = 0\}$$
$$= \phi.$$

Hence, the competence domain of Q with respect to D is empty. Since the domain of D ($= \{s | a[0] \neq 0\}$) is not empty, we infer that Q is not correct with respect to D, hence it has faults. We consider programs Q'_1, Q'_2 and Q' obtained from Q by applying the atomic changes of AC_0, and we compute their competence domains:

$$Q'_1 = \{(s, s') | a[0] = a'[0] \wedge \forall k : 1 \leq k \leq N - 1 : a'[k] = 0 \wedge a[N] = a'[N]\}.$$

$$Q'_1 \cap D = \{(s, s') | a[0] \neq 0 \wedge a[0] = a'[0] \wedge \forall k : 1 \leq k \leq N : a'[k] = 0 \wedge a[N] = a'[N]\}.$$

$$= \{(s, s') | a[0] \neq 0 \wedge a[N] = 0 \wedge a[0] = a'[0] \wedge \forall k : 1 \leq k \leq N : a'[k] = 0\}.$$

Hence, the competence domain of Q'_1 with respect to D is:

$$CD'_1 = \{s | a[0] \neq 0 \wedge a[N] = 0\}.$$

This is a superset of the competence domain of Q with respect to D, hence 0 (in the initialization of i) is a fault in Q with respect to D; because this fault has a single syntactic atom, it is a unitary fault; and because the competence domain of Q'_1 is not equal to the domain of D, Q'_1 is not correct with respect to D. We now consider Q'_2:

$$Q'_2 = \{(s, s') | \forall k : 0 \leq k \leq N : a'[k] = 0\}.$$

$$Q'_2 \cap D = \{(s, s') | a[0] \neq 0 \wedge a'[0] \neq 0 \wedge \forall k : 0 \leq k \leq N : a'[k] = 0\}.$$

$$= \phi.$$

Hence, the competence domain of Q'_2 with respect to D is empty. We now consider Q':

$$Q' = \{(s, s') | a[0] = a'[0] \wedge \forall k : 1 \leq k \leq N : a'[k] = 0\}.$$

Hence,

$$Q' \cap D = \{(s, s') | a[0] \neq 0 \wedge a[0] = a'[0] \wedge \forall k : 1 \leq k \leq N : a'[k] = 0\}.$$

Hence, the competence domain of Q' is:

$$CD' = \{s | a[0] \neq 0\},$$

which is the same as the domain of D, hence Q' is absolutely correct with respect to D. The transformation from Q to Q'_1 is a unitary fault repair but the transformation from Q to Q' is not, hence the fault density of Q is 1. Since Q' is absolutely correct, the fault depth of Q is 2.

If we now consider the same program Q and the same specification D, but the new (larger) set of atomic changes AC_1, we can add a new candidate repair to Q, namely program Q'' obtained from Q by replacing the array reference `a[i]` by `a[i+1]`. We find:

$$Q'' = \{(s, s') | a[0] = a'[0] \wedge \forall k : 1 \leq k \leq N : a'[k] = 0\}.$$

The competence domain of Q'' with respect to D is:

$$CD'' = \{s | a[0] \neq 0\},$$

which is the same as the domain of D, hence Q'' is absolutely correct with respect to D. With the set of atomic changes AC_1, we now have two programs that stem from Q by unitary fault repair (namely, Q'_1 and Q''), hence the fault density of Q with respect to D is 2; its fault depth is 1 since Q'' (obtained from Q by one unitary fault repair) is absolutely correct.

5 A generic algorithm for program repair

Once we define the concepts of absolute correctness and relative correctness, the outlines of a program repair algorithm become clear: enhance relative correctness until absolute correctness is achieved. Also, now that we understand the concept of unitary fault, it may be advantageous, for the sake of controlling combinatorial explosion, to design program repair

algorithms in such a way as to repair one unitary fault at a time; so that if we apply multiple mutations, it is not to repair several faults at once, but rather to repair one fault that has higher (than 1) multiplicity. Hence, we may have to apply several atomic changes at a time, but the number of simultaneous atomic changes is limited by the maximum fault multiplicity we want to consider ($\mu = 2$ or 3 at most, in practice) rather than the program's fault depth (which is usually unknown and unbounded). The goal of this section is to write code for the proposed program repair algorithm, starting with the oracles that test for absolute and relative correctness.

5.1 Absolute correctness

An oracle takes the form of a binary predicate that refers to the initial state and the final state of the program, as in the following generic pattern:

```
{inits = s;  //save the initial state
P(); // under test, modifies s, but preserves inits
assert (oracle(inits,s));}
```

It may be tempting to think that if we are testing a program for absolute correctness with respect to specification R then the oracle takes the form $(s, s') \in R$. But that would be wrong, for the following reason: let s be a test datum outside the domain of R; then regardless of what $P(s)$ is, the predicate $(s, P(s)) \in R$ returns false (since by hypothesis s is not in the domain of R); as a result, P will be deemed incorrect, when in fact it should not be held accountable for its behavior outside the domain of R. Oracle $\Omega(s, s')$, defined below, ensures that P is tested only for inputs that are in the domain of R.

Definition 7 Given a specification R on space S, the *oracle of absolute correctness* derived from R is denoted by $\Omega(s, s')$ and defined by:

$$\Omega(s, s') \equiv (s \in dom(R) \Rightarrow (s, s') \in R).$$

We say that program P satisfies oracle $\Omega(s, s')$ on state s if and only if $\Omega(s, P(s))$ is true. Also, we say that program P satisfies oracle $\Omega(s, s')$ for test suite T if and only if it satisfies the oracle for all elements $s \in T$. The following proposition, where ${}_{T\backslash}R$ represents the pre-restriction of relation R to set T, confirms that this definition is sound.

Proposition 8 *Let $\Omega(s, s')$ be the oracle of absolute correctness derived from specification R on space S and let T be a subset of S. A program P is absolutely correct with respect to ${}_{T\backslash}R$ if and only if execution of P on every element of T satisfies oracle $\Omega(s, s')$.*

The proof of this Proposition is given in "Appendix A". Based on this proposition, we derive the following oracle:

```
bool absoluteCorrectness(testData T) //                    1
   {statetype inits,s;  bool abscor=true; //               2
    while (moretestdata(T)) //                             3
       {inits =gettestdata(T);//load test datum             4
        s = inits;p();// modifies s, preserves inits       5
        abscor= abscor&&absoracle(inits,s);}//             6
    return abscor}//                                       7
bool absoracle(statetype s, sprime)//                      8
   {return (!domR(s) || R(s,sprime))}//re: Definition 7    9
```

We assume that `gettestdata(T)` retrieves in turn all the elements of T at successive calls, until `moretestdata(T)` returns false. For each element s of T, this program saves the initial state in variable `inits` and calls program `p()`, which modifies s but keeps `inits` intact. Upon execution of program `p()` (line 5), the pair `(inits,s)` is ready to be checked for correctness; the Boolean function `absoracle(inits,s)` checks the correctness of P for a single execution and `absoluteCorrectness(T)` checks the correctness of P for all the elements of T; according to Proposition 8, if and only if a program P runs successfully on all the elements of set T, it is absolutely correct with respect to ${}_{T\backslash}R$. We assume that we have at our disposal the unary predicate `domR()` and the binary predicate `R(,)` that represent the specification R and its domain. Line 9 reflects the proposed formula and line 6 cumulates this assertion over set T.

Since we are testing P only on set T, we cannot hope to prove more than the absolute correctness of P with respect to ${}_{T\backslash}R$; Proposition 8 provides that we can prove no less.

5.2 Relative correctness

Given a space S, a specification R on S and a program P on S, we consider the execution of some program P' on some initial state s, and we assume that this execution terminates normally and returns a final state s'. We want to derive an oracle that analyzes the pair of states (s, s') and determines whether it is consistent with the premise that P' is more-correct than P with respect to R. Relative correctness of a program P' over program P with respect to specification R means, according to Definition 3, that wherever program P satisfies specification R (i.e., $\Omega(s, P(s))$ holds), so does the program under test ($\Omega(s, s')$). Hence, the following definition.

Definition 8 Given a specification R on space S and a program P on S, the *oracle of relative correctness* over P with respect to R is denoted by $\omega(s, s')$ and defined by:

$$\omega(s, s') \equiv (\Omega(s, P(s)) \Rightarrow \Omega(s, s')).$$

The following proposition provides that this definition is sound.

Proposition 9 *Let $\omega(s, s')$ be the oracle of relative correctness over program P with respect to specification R and let T be a subset of S. A program P' is more-correct than P with respect to ${}_{T\backslash}R$ if and only if execution of P' on every element of T satisfies oracle $\omega(s, s')$.*

The proof of this Proposition is given in "Appendix B". Based on this proposition, we derive the following oracle of relative correctness over program P with respect to specification R:

```
bool relativeCorrectness(testData T)//                          1
 {statetype inits, s; bool relcor=true;//                       2
  while (mortestdata(T))//                                      3
    {inits = gettestdata();// load test datum                   4
     s = inits; p(); // modifies s, preserves inits             5
     bool abscor = absoracle(inits,s);//                        6
     s = inits; Pprime();//                                     7
     relcor =relcor &&   //                                     8
       (!abscor||absoracle(inits,s));}//re: Definition 8        9
     return relcor}//                                          10
```

According to Definition 8, program `Pprime` is more-correct than program `P` if and only if whenever `Pprime` runs successfully on some element of test suite T, so does `P`. Line 4 generates a test datum into `inits`, and line 5 moves `inits` into `s` and invokes program

P, which modifies state s but keeps inits intact. Line 6 records into variable abscor whether or not execution of P on s was successful, and line 7 reinitializes the state and runs program Pprime on it. Line 9 checks the relative correctness formula of Definition 8 for the current test data and line 8 cumulates the outcome of the current execution with those of past executions. Line 10 returns the combined outcome of all the tests in T. According to Proposition 9, if and only if a program P' runs successfully on all the elements of set T, it is more-correct than P with respect to ${}_{T\backslash}R$.

Since we are testing P' only on set T, we cannot hope to prove more than the relative correctness of P' over P with respect to ${}_{T\backslash}R$; Proposition 9 provides that we can prove no less.

Predicate abscor represents the absolute correctness of P on $inits$ and predicate absoracle(inits,s) represents the absolute correctness of the program under test on the same input. Line 9 represents the formula of $\omega()$ and line 8 cumulates this formula over T.

5.3 Strict relative correctness

We are given a specification R on space S and a program P on S. We consider a program P on S and we want to write an oracle that checks whether a program P' is strictly more-correct than P with respect to R. P' is strictly more-correct than P if and only if it is more-correct than P and there exists at least one input s on which P fails and P' succeeds; whence the following formula.

Definition 9 Given a specification R on space S and a program P on S, the *oracle of strict relative correctness* over P with respect to R is denoted by $\sigma_T(P')$ and defined by:

$$\sigma_T(P') \equiv (\forall s \in T : \omega(s, P'(s))) \wedge (\exists s \in T : \neg\Omega(s, P(s)) \wedge \Omega(s, P'(s))).$$

The following proposition provides that this definition is sound.

Proposition 10 *Let $\sigma_T(P')$ be the oracle of strict relative correctness over program P with respect to specification R and let T be a subset of S. A program P' is strictly more-correct than P with respect to ${}_{T\backslash}R$ if and only if oracle $\sigma_T(P')$ returns true.*

The proof of this Proposition is given in "Appendix C". The test driver for strict relative correctness is written as:

```
bool StrictRelCor(testData T)//                                          1
 {statetype inits, s;//                                                  2
  bool strict,relcor;strict=false;relcor=true;//                         3
  while (mortestdata())//                                                4
    {inits = gettestdata();// load test datum                            5
     s = inits; p(); // modifies s, preserves inits                      6
     bool abscor = absoracle(inits,s);//                                 7
     s = inits; Pprime();//                                              8
     relcor =relcor && //                                                9
            (!abscor || absoracle(inits,s));//                           10
     strict =strict || //                                                11
              (!abscor && absoracle(inits,s))}//re: Definition 9          12
  return (strict && relcor)}//                                           13
```

Line 10 reflects the condition of relative correctness, and line 9 reflects the universal quantification over T; line 12 reflects the condition of strict relative correctness, and line 11 reflects the existential quantification over T.

Program Pprime is strictly more-correct than program P if and only if Pprime is more-correct than P, and there exists at least one instance where Pprime runs successfully and P fails. Hence to test for strict relative correctness, it suffices to test for relative correctness, and check if any test datum causes P to fail while Pprime succeeds; this is exactly what lines 11 and 12 are keeping track of.

Since we are testing P' only on set T, we cannot hope to prove more than the strict relative correctness of P' over P with respect to $_{T\setminus}R$; Proposition 10 provides that we can prove no less.

5.4 Unitary fault repair

The oracles derived in the previous sections give us means to recognize valid repairs, but do not give us means to generate valid repairs. Hence, all we can do for now is to derive an algorithm that, given a function to generate plausible repair candidates, can select those that are indeed valid repairs, i.e., are strictly more-correct than the original. The purpose of this section is to write code that performs a unitary increment of relative correctness; as we have seen in Sect. 4.4, this amounts to repairing a unitary fault. To this effect, we assume the availability of a patch generator that applies a set of atomic change operators, and we seek to enhance correctness by attempting to identify faults of increasing multiplicity, starting at 1, and not exceeding a user-specified maximum, M. For the sake of argument, we assume that candidate patches are organized as a set of patch streams of increasing multiplicities, which we name, respectively, $PS(1)$, $PS(2)$,..., $PS(M)$. Each patch stream $PS(m)$ is an ordered sequence, to which we may apply sequence operators *head()* and *tail()*, referring, respectively, to the first element, and the remainder of the sequence. We assume that the patch generator puts at our disposal the following functions:

- MorePatches(P,m), a Boolean function that returns true if and only if there remains more patches of P of multiplicity m.
- NextPatch(P,m), which returns the next element of $PS(m)$, for $1 \leq m \leq M$.

Using these functions, and assuming that R (the specification) and T (the test data set) are global variables, we write the following code:

```
void UnitIncCor(programType P, int M,                                  //1
              bool& inc, programType& Pp)                              //2
 {//attempts to return in Pp a correctness enhancement of P,          //3
  //using m atomic changes, 1<=m<=M                                   //4
  //inc=true iff Pp strictly more-correct than P.                     //5
 int m=1; inc=false; Pp=P;                                           //6
 while (not inc && m<=M)                                             //7
   {//increase correctness with m changes                             //8
   while (not smc(Pp,P) && MorePatches(P,m))                         //9
     {//smc: Pp strictly more-correct than P                         //10
     Pp = NextPatch(P,m);}                                           //11
   if smc(Pp,P) {inc=true;}                                          //12
   else {m=m+1;}}//try higher multiplicity                           //13
 }                                                                   //14
```

This algorithm is written in C-like pseudo-code, though it takes great liberties with its syntax (assuming the existence of a programType data type, using programs as parameters, etc.). This algorithm takes as input a program P and maximum multiplicity M and attempts to return a program P' (Pp) that is strictly more-correct than P without exceeding M atomic changes to P. Variable inc tells whether a strictly more-correct program was found.

5.5 Stepwise correctness enhancement

Now that we have a routine to perform unitary increments of correctness enhancement, we use it to repair program P by applying this routine repeatedly until we find an absolutely correct program or (due to the imperfection of patch generation) we stall before we reach an absolutely correct program (i.e., we find a program that is not absolutely correct but whose correctness we cannot enhance).

```
void ProgramRepair(programType& P,              //1
         specification R, testdata T, int M)    //2
{bool inc=true;                                 //3
 while (inc && not abscor(P))                   //4
   {UnitIncCor(P, M, inc, Pp);                  //5
   if inc {P=Pp;}}}                             //6
```

The while loop iterates as long as we keep incrementing the correctness of the program, and it is not yet absolutely correct. In line 5, we attempt to place in Pp a (stepwise) repair of P, and if successful, we set inc to true. In line 6, if we find that we have enhanced correctness, then we let P be the enhanced program, and we check for absolute correctness, else resume correctness enhancement on P.

6 Algorithm analysis

In this section, we analyze the functional and performance attributes of the program repair algorithm proposed in the previous section, using analytical and empirical means.

6.1 Big Oh complexity

To estimate the *Big Oh* complexity of `UnitIncCor()`, we assume that the patch generation has a fixed fan-out for program P, say $F(P)$, when the multiplicity is 1; we also assume that for a greater multiplicity, say m, the fan-out is $F(P)^m$. Then, the *Big Oh* complexity of `UnitIncCor()` is bounded by the following formula: $O(F(P)^M \times |T|)$. Indeed, this algorithm is made up of two nested loops: the number of iterations of the innermost loop is bounded by $F(P)^M$ and each iteration runs $|T|$ tests to check for strict relative correctness of Pp over P; the number of iterations of the outer loop is bounded by $F(P)^M$. As for the *Big Oh* performance of `ProgramRepair()`, it depends on the number of iterations of the outer loop, which is unbounded; though we would like to think that the number of iterations of this loop is the fault depth of the program, we have no way to ensure that. The fault depth of the program is the minimal number of unitary fault removals that separate the program from correctness; we have no way to ensure that the algorithm follows a minimal path.

6.2 Correctness properties

In this section, we use Hoare logic [25, 52] to prove partial correctness and termination properties of the proposed algorithm with respect to specifications formulated as (precondition/postcondition) pairs. The formulation of these specifications as well as the conduct of the proofs are made possible by the vocabulary of concepts of the proposed theory.

6.2.1 Specifications

For the sake of simplicity, we focus in this paper on the function that performs a unitary increment of correctness, i.e., `UnitIncCor()`; the generic algorithm is merely the iterative application of this function until correctness enhancement ends, either because we have reached absolute correctness, or because patch generation is unable to generate a valid patch (i.e., a mutant that is strictly more-correct than P) within the maximum multiplicity (M). We can consider any number of specifications for `UnitIncCor()`; we present a sample below, where we represent ${}_{T\setminus}R$ by R'. All claims of correctness (absolute or relative) are implicitly understood to be with respect to R'.

- `UnitIncCor()` does no harm.
  - Precondition: **true** .
  - Postcondition: $Pp \sqsupseteq_{R'} P$.
- If the patch generator can generate a strictly more-correct program, then `UnitIncCor()` will enhance correctness.
  - Precondition: $(\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P)$.
  - Postcondition: $Pp \sqsupset_{R'} P$.
- If upon execution of `UnitIncCor()` inc is **true** , then Pp is strictly more-correct than P with respect to R'.
  - Precondition **true** .
  - Postcondition: $(inc \Rightarrow Pp \sqsupset_{R'} P)$.

6.2.2 Partial correctness: perfect recall

Proposition 11 *Function* `UnitIncCor()` *has perfect recall, in the sense that if the patch stream has a program that is strictly more-correct than P, then* `UnitIncCor()` *will return in Pp a program that is strictly more-correct than P.*

Proof We must prove that the following formula is valid in Hoare's deductive logic [25].
v: $\{(\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P)\}$

```
m=1; inc=false; Pp=P;
while (! inc && m<=M)
  {while (! smc(Pp,P) && MorePatches(P,m))
    {Pp = NextPatch(P,m);}
   if smc(Pp,P) {inc=true;}
   else {m=m+1;}}//try higher multiplicity
```

$\{Pp \sqsupset_{R'} P\}$.

The proof of this formula is given in the appendix, Sect. D. □

Note that `UnitIncCor()` does not retrieve all the patches that are strictly more-correct than P; it only retrieves the first patch that it encounters. Hence, all we can say is that if there exists a patch Q in the patch stream that is strictly more-correct than P, then `UnitIncCor()` will necessarily return in Pp a program that is strictly more-correct than P (this could be Q or it could be another patch that it encounters before Q).

6.2.3 Partial correctness: perfect precision

Proposition 12 *Function* `UnitIncCor()` *has perfect precision, in the sense that if* inc *is set to* **true** *then* Pp *is strictly more-correct than* P.

Proof We must prove the validity of the following formula in Hoare logic [25]:
v: {**true** }

```
m=1; inc=false; Pp=P;
while (! inc && m<=M)
  {while (! smc(Pp,P) && MorePatches(P,m))
    {Pp = NextPatch(P,m);}
   if smc(Pp,P) {inc=true;}
   else {m=m+1;}}//try higher multiplicity
```

$$\{inc \Rightarrow Pp \sqsupset_{R'} P\}.$$

The proof of this formula is given in the appendix, Sect. E. □

Note that Pp is initialized to P and it is modified only if the patch stream includes a program that is strictly more-correct than P; in that case, inc is set to true.

6.2.4 Termination

A while loop of the form `w: {while (t) {b}}` on space S can be proved to terminate for precondition **true** if we can prove that the transitive closure of $(T \cap B)$ is well founded, i.e., it admits no infinite chains. This, in turn, can be proved by finding a superset of the transitive closure of $(T \cap B)$ that is well founded (any transitive superset of $(T \cap B)$ is a superset of the transitive closure of $(T \cap B)$). One way to do so is to find a *variant function* f on space S that satisfies the following condition:

$$t(s) \wedge s \in dom(B) \Rightarrow f(B(s)) < f(s),$$

for some well-founded relation $<$ on S.

In light of this, we find that function `UnitIncCor()` terminates for precondition **true** for the following reason: the inner loop can be proved to terminate using the function $f() = length(PS(m))$, which decreases by 1 at each iteration and is bounded by 0. The outer loop can be proved to terminate using the function $g() = \langle M - m, inc \rangle$, where each iteration either increases m (hence decreases $M - m$) or preserves m but decreases inc (if we consider **true** $<$ **false**).

As for function `ProgramRepair()`, we have no assurance that it terminates in general, but we do have a simple sufficient condition: if $dom(R)$ is finite then each iteration of the loop of `ProgramRepair()` reduces the function

$$f() = dom(R) \setminus dom(R \cap P),$$

by at least one element, and is bounded by the empty set.

7 Instances of the generic algorithm

The generic algorithm is not a program repair tool; it is merely a template for repairing programs by relative correctness, assuming we are provided the patch generation functions `NextPatch(P,m)` and `MorePatches(P,m)`. The algorithm is generic in the sense that

it specifies how candidate patches are selected/ validated, but does not specify how they are generated, hence we can create an instance thereof for each patch generator. In this section, we present two instances of this algorithm, which use patch generators from two existing tools:

- *GRCFix* (*GenProg-based Relative Correctness Fix*), an instance of the generic algorithm derived from GenProg's patch generator [40, 41, 43]. This is the subject of Sect. 7.1.
- *CRCFix* (*Cardumen-based Relative Correctness Fix*), an instance of the generic algorithm derived from Cardumen's patch generator [56]. This is the subject of Sect. 7.2.

For each experiment, we present the following data in tabular form:

- *Unitary Faults* in the order in which they are repaired by our algorithm.
- *Mutations that make up the unitary fault.* Whereas `UnitIncCor()` repairs one unitary fault at a time, this may involve more than one mutation since some unitary faults have higher than 1 multiplicity. We may, in such cases, add a column in which we specify the value of m that was set by `UnitIncCor()`.
- *Number of Passing Tests.* These are test data where the program behaves correctly; this number typically increases with each fault repair.
- *Number of Failing Tests.* These are test data where the program fails; this number typically decreases with each fault repair.
- *Repair.* In this column we present brief details about the execution of `UnitIncCor()` on each fault.
- *Number of Variants.* In this column, we report on the number of variants that `UnitCorInc()` inspected before finding a valid repair.
- *Execution Time in seconds*.
- *Outcome of the Original Program Repair Tool.* In this column, we report on the outcome of running the original program repair tool (jGenProg, Cardumen) on the same benchmark program seeded with the indicated benchmark mutations. Note that the isolation of faults, and their successive repairs, is the result of executing the generic algorithm in its various instances (GRCFix, CRCFix), hence in the last column (which reports on jGenProg and Cardumen) we merge the rows, and show the outcome of executing the original program repair tools on the program seeded simultaneously with all the mutations.

7.1 GRCFix: implementation

We consider *Astor* [54, 55], a program repair framework that contains five Java-specific tools (jGenProg, jKali, jMutRepair, DeepRepair and Cardumen) and offers researchers/ developers the ability to override, replace or combine their functionality to produce new tools. Specifically, Astor provides twelve extension points that form the design space of program repair; novel repair approaches can be implemented by choosing an original component for each extension point. For the purposes of our research, we choose to adapt jGenProg as follows:

- *Fault Localization*: Astor/jGenProg uses an existing fault localization technique called GZoltar [1].
- *Scale*: jGenProg analyzes programs at the statement level.
- *Patch Generation*: we adopt three types of atomic changes: removing, inserting, and replacing statements.
- *Navigation*: For the sake of recall, we adopt an exhaustive navigation approach. By applying function `UnitIncCor()`, which seeks to performs a unitary increment of

correctness, we ensure that we repair one unitary fault at a time, thereby controlling the combinatorics of the search space.
– *Patch Validation*: Astor provides an extension point named *EP PV*, to specify the patch validation process. We adopt the patch validation depicted in Sect. 5.

We execute GRCFix on programs on which we have applied several changes; these may correspond to multiple faults, or to fewer (than the number of changes) faults with higher (than 1) multiplicities. GRCFix repairs them one (unitary) fault at a time, producing increasingly more-correct programs, until it achieves absolute correctness or stalls (is unable to enhance correctness). The outer loop of GRCFix (re: `ProgramRepair()`) repairs several faults, one at a time, by calling `UnitIncCor()` iteratively; `UnitIncCor()` repairs a single fault by attempting single atomic changes, else double atomic changes, else triple atomic changes, etc., until it achieves strict correctness enhancement or reaches the maximum allowed multiplicity (M).

We consider the *Math* project of *Defects4J* [30], and we run four experiments that consist in seeding the program with several changes then deploying *GRCFix* and showing its performance; we refer the interested reader to [32] for more experiments.

7.1.1 Math70, Math73, Math95

For this experiment, we select faults *Math70*, *Math73* and *Math90*; for each fault we show the original (faulty) code and the patch that GRCFix identifies for it (using JGenProg's patch generator).

– *Math70*. Fault: `return solve(min,max).` Patch: `return solve(f,min,max).`
– *Math73*. Fault: `return solve(f,min,yMin,max,yMax,initial,yInitial).` Patch: `return solve(f, min, max).`
– *MathI2*. Fault: `double real = 4.0*real;`. Patch: `double real = 2.0*real;`.

We deploy GRCFix with the following parameters:

– Fault localization threshold: 0.5.
– Number of modification points: 12.
– Maximum multiplicity of unitary faults, M: 1.
– Maximum execution time: 9 h.

The outcome of this experiment is summarized in Table 2, where the faults are listed in the order in which they are repaired by GRCFix.

All three faults were repaired within 14 min; each fault was repaired by a single call to `UnitCorInc()`, with m=1. In this experiment, the fault density and the fault depth of the program were 3, and all faults had a multiplicity of 1. Exceptionally in this case, we start with 3 failing tests, and decrement by 1 after each fault repair; generally we advocate for a much larger test suite, including a larger set of failing tests.

Table 2 Benchmark mutations: Math70, Math73, Math95

Faults	Mutations	Pass tests	Fail tests	Repair operations	Number of variants	Time (s)	JGenProg
Fault1	Math70	3599	3	Return solve (f,min,max)	1	24	
Fault2	Math73	3600	2	Return solve (f, min, max)	1	23	Timeout
Fault3	Math95	3601	1	-ret = d/(d − 2.0) + ret = 0.0	112	797	

We run jGenProg on the same program seeded with all three faults, using the following parameters:

- Maximum number of generations: 500.
- Population size: 40.
- Fault localization threshold: 0.1.
- Maximum Execution time: 9 h.

Execution of jGenProg times out after 9 h without success; inspection of the variants shows that it did generate one of the patches that enhances correctness, but since it is testing for absolute correctness, that variant was discarded. Whenever one fault is repaired, GRCFix uses the new repaired program as the base, runs fault localization on the new program, and is poised to target the next fault with greater precision.

7.1.2 Math50, Math53, MathI

The selected faults and their patches are:

- *Math50*. Fault: `if(x==x1) {x0=0.5*(x0+x1- FastMath.max (rtol*FastMath.abs(x1), atol)); f0=computeObjectiveValue(x0);}`. Patch: Code deletion.
- *Math53*. Fault: A Missing Statement.
  Patch: `if (isNaN|| rhs.isNaN){return NaN;}`.
- *MathI*. Fault: `d=FastMath.cos(real2)- MathUtils.cosh (imaginary2);`.
  Patch: Code deletion.

We deploy GRCFix with the same parameters as the first experiment. The results of the experiment are summarized in Table 3.

All three faults are removed within approximately two and a half hours, in the order shown in Table 3. As in the previous example, the number of failing tests decreases with each fault repair, and as in the previous example, we are looking at a program with fault density of 3, fault depth of 3, and fault multiplicity of 1 for each fault.

We run jGenProg on the same program seeded with all three faults, using the following parameters:

- Maximum number of generations: 500.
- Population size: 40.
- Fault localization threshold: 0.5.
- Maximum execution time: 9 h.

Execution of jGenProg times out after 9 h without success. We try running jGenProg again by changing several of its parameters, increasing the number of generations to ten million, to no avail. The *Defects4J* benchmark has a prespecified patch for each of its faults; in two cases in our experimentation (Math95 in the first experiment and Math50 in the second experiment)

Table 3 Benchmark mutations: Math50, Math53, MathI

Faults	Mutations	Pass tests	Fail tests	Repair operations	Number of variants	Time (s)	JGenProg
Fault1	MathI	3598	4	Code deletion	329	7245	Timeout
Fault2	Math50	3599	3	Code deletion	2	428	
Fault3	Math53	3600	2	Missing statement	24	1785	

the patch generated by GRCFix is not the same as the patch listed by Defects4J. There are two reasons why this may arise:

- What is listed as a patch in the Defects4J benchmark is just the original code that was in the program before the fault was seeded; it is *one possible patch*, it is not *the only patch*.
- GRCFix is guaranteed, by Proposition 8, to produce a program that is absolutely correct with respect to $R' =_{T\setminus} R$; it may still be incorrect with respect to R.

7.1.3 Complex1, Complex2

Whereas the cases we have seen so far (in [32] and in the examples above) involve single-site faults, this and the next case involve multi-site unitary faults; in such cases, we need to perform more than one atomic change before we can achieve strict relative correctness. Below are the changes we introduced, along with their repairs.

- *Complex1*: Fault: `double real2 = 2.0*real;`
  Patch: `double real4=4.0*real; double imaginary2=2.0*imaginary; double d = Math.cos(real2)+MathUtils.cosh(imaginary2);`
- *Complex2*: Missing statement.
  Patch: `d = Math.cos(real2)*2.0 *MathUtils.cosh(imaginary2); return createComplex(Math.sin(real2)/d,MathUtils.sinh(imaginary2)/d);`

GRCFix calls `UnitIncCor()` which generates 462 repair candidates through single mutation, and finds that none of them enhances (strict relative) correctness; it then considers these mutants in turn and generates mutants thereof; the first mutant yields 462 secondary mutants, none of which is strictly more-correct than the original program; the second was used in a double mutation, and the 344th mutant thereof proved to be strictly more-correct than the original, and is in fact absolutely correct. The repair concludes successfully in less than 46 min; we are dealing here with a single two-site fault, since it stems from two mutations, none of which was found to enhance correctness by itself. Table 4 summarizes the outcome of this experiment.

We run jGenProg on the same program with the same mutations using the parameters discussed in the previous sections; it times out after 9 h of execution time, without repairing any fault.

Table 4 Benchmark mutations: Complex1, Complex2

Faults	Mutations	m =	Pass tests	Fail tests	Repair operations	Number of variants	Time (s)	JGenProg
Fault 1	Complex1 and Complex2	1	3600	2	Single mutation	462	962	Timeout
Fault 1	Complex1 and Complex2	2	3600	2	Double mutation 1	462	1065	
Fault 1	Complex1 and Complex2	2	3600	2	Double mutation 2	334	712	

Table 5 Benchmark mutations: Complex1, Complex2, Math70, Math73

Faults	Mutations	m =	Pass tests	Fail tests	Repair operations	Number of variants	Time (s)	JGenProg
Fault 1	Complex1 and Com-plex2	2	3598	4	Double mutation	1268	2739	
Fault 2	Math 70	1	3600	2	Return solve (f,min,max)	1	24	Timeout
Fault 3	Math 73	1	3601	1	Return solve (f, min, max)	1	23	

7.1.4 Complex1, Complex2, Math70, Math73

Whereas the first two examples illustrate the operation of GRCFix on programs with greater fault depth, and the third example illustrates the operation of GRCFix on faults with greater fault multiplicity, this example illustrates the operation of GRCFix with greater fault depth and greater fault multiplicity. We seed the benchmark program with four mutations: Complex1, Complex2, Math70 and Math73. the outcome of the experiment is shown in Table 5.

The four changes represent three faults: a two-site fault and two single-site faults. GRCFix repairs the two-site fault in less than 46 min then repairs the other two in less than a minute. Note that we perform fault localization after each fault repair, so that each fault repair helps us target the next fault with greater precision.

We run jGenProg on the same program with the same mutations using the parameters discussed in the previous sections; it times out after 9 h of execution time, without repairing any fault.

7.2 CRCFix: implementation

CRCFix is an instance of the generic algorithm, using the patch generation functionality of *Cardumen* [56], which is an element of the *Astor* framework of Martinez and Monperrus [55]. The CRCFix tool can be accessed at [4] and the data presented in this section can be accessed at [3]; the procedure we followed to derive CRCFix is the same as that depicted in Sect. 7.1, substituting Cardumen for GenProg.

Whereas in Sect. 7.1 we used benchmark programs and benchmark mutations (aka faults), in this section we use a program taken from [44], and some mutations taken from the same source, but we use different experimental parameters, including:

– *A Different Specification.* Whereas the benchmark uses the unseeded program as the specification against which repair is sought, we derive our own specification, ensuring that the unseeded program is correct with respect to it.
– *A Non-Deterministic Specification.* We purposefully select a non-deterministic specification, so as to illustrate how one can repair a program by preserving correctness without preserving correct behavior; in other words, the repaired program is more-correct than the original program with respect to the specification, even though it does not imitate the correct behavior of the original program. This is possible only if correct behavior is not unique, whence the need for non-determinacy.

– *Arbitrary Test Data.* Whereas benchmark data comes with a fixed test suite, we generate our own, so as to be able to illustrate the (positive) impact of larger test suites on the efficiency and precision of the repair operation.

7.2.1 The program and specification

We consider a binary search program on space S defined by an array a, a variable x, and an index k. The specification is:

$$R = \{(s, s')|sorted(a) \wedge a[k'] = x\}.$$

The domain of this relation is:

$$dom(R) = \{s|sorted(a) \wedge x \in a\}.$$

R and $dom(R)$ are represented by, respectively, a binary predicate and unary predicate, which we use to derive oracles as per definitions 7, 8 and 9. The program, due to [44], is written as:

```
{int lo=0; int hi=a.length; int k=-1;      //1
while (lo<hi)                              //2
   {int mid=(lo+hi)/2;                     //3
    if (x==a[mid]&&(mid==0||x!=a[mid-1]))//4
       {k=mid; break;}                     //5
    else                                   //6
       if (x<=a[mid]) {hi=mid;}            //7
       else {lo=mid=1;}}                   //8
return k;}                                 //9
```

Some of the modifications we seed are taken from the *QuixBugs* benchmark [44], although we seed more than one modification at a time, and we use R as the specification; we use the test data provided in the benchmark, though we may enlarge it as needed. Unless we specify otherwise, Cardumen is limited to 500 generations and a run-time of 120 min.

7.2.2 Two modifications

In this example, we make two modifications to the code:

– Mod 1: `k=mid` in line 5 → `k=mid-1`.
– Mod 2: `hi=mid` in line 7 → `hi-hi-1`.

Table 6 Two faults, Mod1 and Mod2

Faults	Mutations	Pass tests	Fail tests	Repair operations	Number of generations	Time (s)	Cardumen
Fault 1	Mod 1	3	4	k = hi + k	24	59	Aborted
Fault 2	Mod 1	4	3	k = hi + k → k = (mid + mid)/2	104	516	Max generations
Fault 3	Mod 2	5	2	hi = hi + k	24	91	

Table 7 Preserving correctness

Faults	Mutations	Pass tests	Fail tests	Repair operations	Number of generations	Time (s)	Cardumen
Fault 1	Mod 2	5	2	hi = hi-1	25	121	Aborted max generations

The outcome of this experiment is summarized in Table 6. To repair this program, CRCFix went through three iterations of `UnitIncCor()`: Two iterations are needed to repair Mod1 and one to repair Mod2. The whole operation takes about 11 min and uses no more than 104 generations.

When we execute Cardumen on the same data, it runs for 1090s (18 min) and aborts, because it has reached the maximum number of generations allowed (500).

7.2.3 Preserving correctness

In this example, we show how we can preserve correctness without preserving correct behavior, since correct behavior is not unique. We make two modifications:

- Mod 1: `mid=(lo+hi)/2` in line 3 → `mid=h-1`.
- Mod 2: `hi=mid` in line 7 → `hi-mid-1`.

The outcome of this experiment is given in Table 7. This example is interesting because even though we applied two mutations, Mod1 and Mod2, CRCFix made the program absolutely correct with respect to $R' =_{T\setminus} R$ by repairing Mod2 alone, hence the fault depth of this program is 1. Whether its fault density is 2 depends on whether Mod1 is a fault, which we do not know. The whole repair operation took 121 s (2 min) and 25 generations.

When we execute Cardumen on the same data, it runs for 1338s (22 min) and aborts, because it has reached the maximum number of generations allowed (500).

7.2.4 Non-determinacy

In this example, we illustrate how the use of non-deterministic specifications enables us to enhance recall, by comparison with Cardumen.

- Mod 1: `k=mid` in line 5 → `k=mid-1`.
- Mod 2: `return k` in line 7 → `return k-1`.

The outcome of this experiment is summarized in Fig. 8. Three calls to `UnitIncCor()` were required to repair this program; the operation took a total of 642 s (11 min) and the no more than 159 generation for any call. It produces a program that is absolutely correct with respect to $R' =_{T\setminus} R$, where T has a mere 7 elements. Because the size of T is so small, absolute correctness with respect to ${}_{T\setminus}R$ does not mean much, as it leaves much scope for a program to be correct with respect to ${}_{T\setminus}R$ but still incorrect with respect to R; this is why in the next experiment we will use a large test suite.

When we execute Cardumen on the same data, it runs for 1384s (23 min) and aborts, because it has reached the maximum number of generations allowed (500).

7.2.5 Large data set

In this experiment, we show how larger test suites enhance precision. To this effect, we seed the program with a single mutation, and attempt to repair it with a small test suite, then

Table 8 Non-determinacy

Faults	Mutations	Pass tests	Fail tests	Repair Operations	Number of generations	Time (s)	Cardumen
Fault 1	Mod 1	4	3	k = mid + mid	3	10	Aborted
Fault 2	Mod 1	5	2	k = mid + mid → k = (hi + hi)/2	34	90	Max generations
Fault 3	Mod1	6	1	k = mid = 1	159	542	

Table 9 Large data set

Faults	Mutations	Pass tests	Fail tests	Repair operations	Number of generations	Time (s)	Cardumen
Fault 1	Mod 1	6	1	mid = (hi + lo)/2 → mid = k + hi	195	747	x <= a[mid − 1] → x < hi) 484generations 1400(S)

Table 10 Larger data set

Faults	Mutations	Pass tests	Fail tests	Repair operations	Number of generations	Time (s)	Cardumen
Fault 1	Mod 1	92	8	$(x <= a[(lo + hi)/2])$	19	82	Aborted max generations

increase the size of the test suite and observe its impact. The mutation we choose for this experiment is:

– Mod 1: `x <= a[mid]` in line 7 → `x <= a[mid − 1]`.

The outcome of experiment for a test suite of 7 elements is summarized in Table 9. Both CRCFix and Cardumen find a patch that satisfies the specification $R' =_{T\setminus} R$, but, interestingly:

– Whereas the patch found by CRCFix is correct with respect to R, that found by Cardumen is not correct with respect to R, even though it is correct with respect to R'.
– CRCFix finds its patch after 195 generations, whereas Cardumen requires 484 generations.
– CRCFix completes the operation in 747 s (12 min) whereas cardumen requires 1400 s (23 min).

When we augment the test suite size to 100 (92 pass and 8 fail) we find the results that are summarized in Table 10. CRCFix recovers the original program, which is correct, and does so in 82 s after 19 generations. Interestingly, when we increase the size of T from 7 to 100, CRCFix converges faster, with fewer generations, and produces a better solution; the solution it produces is in fact absolutely correct with respect to R.

When we deploy it on the same data, Cardumen runs for 2759 s (46 min) and aborts because it reaches its maximum allowed number of generations (700 in this case).

8 Conclusion: Summary, assessment and prospects

8.1 Summary

This paper proposes a theory of program repair based on the following premises:

- To repair a program does not necessarily mean to make it absolutely correct; rather it means to make it more-correct than it was originally. Hence, any theory of program repair ought to be based on some concept of relative correctness.
- Relative correctness ought to play for program repair the role that absolute correctness plays for program derivation.
- Any definition of relative correctness ought to satisfy some litmus tests, including reflexivity and transitivity, culmination in absolute correctness and being a sufficient condition for greater reliability.
- In order to talk about a fault, we need to specify four parameters: a specification, a definition of relative correctness, a definition of syntactic atoms, and a set of atomic changes. In order to talk about multiple faults (as in: program P has N faults), we need to specify a fifth parameter: the concept of unitary fault.
- For the sake of effectiveness, program repair tools ought to be designed in such a way as to make provisions for multi-site faults; at the same time, for the sake of efficiency, they ought to be designed in such a way that multiple mutations are attempted only to repair multi-site faults, not to repair multiple faults simultaneously.

We argue that program repair is any transformation that results in a strict enhancement of the program's relative correctness, and we present a generic program repair algorithm whose gist is to enhance relative correctness repeatedly until it achieves absolute correctness.

Using the ontology provided by our theory, we prove partial correctness and termination properties of the generic algorithm in Hoare's deductive logic [25]. Also, using the patch generation capability of existing program repair tools (GRCFix, based on GenProg [43] and CRCFix, based on Cardumen [56]), we derive instances of the generic algorithm, whose performance we showcase using standard benchmark data pertaining to programs and faults. To highlight the capability of GRCFix and CRCFix to handle programs with higher (than 1) fault depth and faults with higher (than 1) fault multiplicity, we deploy them on combinations of mutations provided in the *Defects4J* benchmark and original faults of varying multiplicity and complexity.

8.2 Assessment

The approach advocated in this paper offers a number of novelties with respect to current program repair practice (to the extent that we understand it):

- Whereas most existing program repair tools are validated by showcasing their performance on shared benchmarks of programs and faults, we validate the generic algorithm by means of formal proofs using Hoare logic. We also illustrate the performance of GRCFix and CRCFix on benchmark programs and faults, but we do so as a complement to/confirmation of the static verification, not as a substitute thereto.
- Whereas most existing program repair tools use patch validation criteria that are prone to cause loss of recall and loss of precision, we use a precise criterion for patch validation, based on relative correctness, and we prove that this criterion provides us perfect precision and perfect recall: If patch generation produces at least one patch that is strictly more-

correct than P then the algorithm is guaranteed to strictly enhance relative correctness; and if inc holds on output then program P' is assured to be strictly more-correct than P.
– The generic algorithm (and its instantiations) distinguishes between multi-site faults and multiple single-site faults, and is designed to work at higher (than 1) fault depth and fault multiplicity; it is fine-tuned to maximize effectiveness while controlling combinatorics.

We feel that none of these novelties would be possible without a sound theoretical foundation.

We view our paper as throwing a challenge to researchers in program repair, by encouraging them to specify explicitly what definition of relative correctness their method supports (assuming that they subscribe to our premise that to repair a program means to make it more-correct) and to prove that their method/tool enhances relative correctness as defined. We do not claim that our definition of relative correctness is the only possible definition, nor the best one; in the same way that the study of program derivation has given rise to several distinct definitions of refinement [2, 22, 24, 26, 61], it is possible (even desirable) to envision different definitions of relative correctness for different purposes or contexts.

8.3 Threats to validity

The main threats to validity are the usual concerns that arise with any attempt to formalize software engineering processes, namely:

– *Scalability*. We argue that relative correctness scales as much as, or as little as, absolute correctness; and it plays for program repair the same role that absolute correctness plays for program derivation. One way we envision to deal with scalability is to use a general definition of relative correctness, which applies to non-deterministic programs; this definition, due to Desharnais et al. [13, 14], enables us to reason about programs without capturing all their functional details.
– *The Need to Provide Specifications*. In the same way that specifications are needed to reason about absolute correctness, they are needed to reason about relative correctness. They are an integral part of the bargain that one strikes to achieve greater precision in the claims one makes about software engineering processes and artifacts.

Also noteworthy is the fact that this approach is focused exclusively on program semantics, hence it aims to generate programs that are more-correct than the original, but does not concern itself with whether the repair is considered *good* by professional programmers/software engineers. To integrate such concerns into the repair process requires that we define an ordering between repairs, based on syntactic criteria, and that we alter the repair algorithm to favor higher ranked repairs; this is well beyond the scope of this paper.

8.4 Prospects

Our short term goal is to delpoy our generic algorithm in conjunction with other (than GenProg, Cardumen) patch generation methods, borrowed from existing program repair tools, and to analyze the impact on the performance of these tools. Our medium term goal is to design an original program repair tool that integrates relative correctness concerns into the patch generation step. Our long term goal is to explore techniques for the generation of more-correct-by-design programs in support of program repair, similar to the techniques that were developed in the past for the generation of correct-by-design programs [16, 22, 24, 61], in support of program derivation.

Our goal is to encourage researchers in program repair to specify explicitly what definition of relative correctness they are pursuing (assuming they agree with our premises), and to validate their methods/tools by proving that they enhance relative correctness as defined (the way we do in Sect. 6.2).

Acknowledgements The authors are very grateful to the anonymous reviewers for their valuable feedback, which has greatly enhanced the presentation and content of our paper. This work is partially supported by the NSF under grant number DGE1565478.

A Proof of Proposition 8

Proof *Proof of Sufficiency*. Program P satisfies oracle $\Omega(s, s')$ for test suite T if and only if:

$$\forall s \in T : \Omega(s, P(s)).$$

By the definition of $\Omega(s, s')$, we can rewrite this as:

$$\forall s \in T : s \in dom(R) \Rightarrow (s, P(s)) \in R.$$

Distributing the clause $(s \in T)$, we write:

$$\forall s : s \in T \wedge s \in dom(R) \Rightarrow s \in T \wedge (s, P(s)) \in R.$$

By set theory, we write the left hand side as:

$$\forall s : s \in (T \cap dom(R)) \Rightarrow s \in T \wedge (s, P(s)) \in R.$$

According to the definitions given in Sect. 3.1 we can write $T \cap dom(R) = dom({}_T\backslash R)$, hence:

$$\forall s : s \in (dom({}_T\backslash R)) \Rightarrow s \in T \wedge (s, P(s)) \in R.$$

Since $(s, P(s))$ is also, by definition, an element of P, this can be written as:

$$\forall s : s \in dom({}_T\backslash R) \Rightarrow s \in T \wedge (s, P(s)) \in (R \cap P).$$

If we now view T as a vector rather than a set, we can rewrite

$$\forall s : s \in dom({}_T\backslash R) \Rightarrow (s, P(s)) \in T \wedge (s, P(s)) \in (R \cap P).$$

Taking the intersection, and using associativity, we find:

$$\forall s : s \in dom({}_T\backslash R) \Rightarrow (s, P(s)) \in ((T \cap R) \cap P).$$

Rewriting $(T \cap R)$ as the pre-restriction of R to T, we find:

$$\forall s : s \in dom({}_T\backslash R) \Rightarrow (s, P(s)) \in (({}_T\backslash R) \cap P).$$

From the right hand side, we infer that s is in the domain of $({}_T\backslash R \cap P)$:

$$\forall s : s \in dom({}_T\backslash R) \Rightarrow s \in dom(({}_T\backslash R) \cap P).$$

Since this is true for all s, we write:

$$dom({}_T\backslash R) \subseteq dom(({}_T\backslash R) \cap P),$$

which we rewrite as:

$${}_T\backslash RL \subseteq (({}_T\backslash R) \cap P)L.$$

Given that the inverse inclusion is a tautology, we find:

$${}_T\backslash RL = (({}_T\backslash R) \cap P)L.$$

Hence, by proposition 2, P is absolutely correct with respect to ${}_T\backslash R$.

Proof of Necessity. If P is correct with respect to ${}_T\backslash R$, then by proposition 2 ${}_T\backslash RL \subseteq ({}_T\backslash R \cap P)L$. Interpreting this formula in logical terms, we find:

$$\forall s : s \in dom({}_T\backslash R) \Rightarrow s \in dom({}_T\backslash R \cap P).$$

If this formula holds for all s in S, it holds a fortiori for all s in T:

$$\forall s \in T : s \in dom({}_T\backslash R) \Rightarrow s \in dom({}_T\backslash R \cap P).$$

Because ${}_T\backslash R$ can be written as $T \cap R$, where T is reinterpreted as a vector, because $dom({}_T\backslash R) = T \cap dom(R)$, we can write:

$$\forall s \in T : s \in T \cap dom(R) \Rightarrow s \in T \cap dom(R \cap P).$$

Isolating the clause $(s \in T)$, we get:

$$\forall s \in T : s \in T \wedge s \in dom(R) \Rightarrow s \in T \wedge s \in dom(R \cap P).$$

Removing the clause $(s \in T)$, which is now redundant, we find:

$$\forall s \in T : s \in dom(R) \Rightarrow s \in dom(R \cap P).$$

Since P is deterministic,

$$\forall s \in T : s \in dom(R) \Rightarrow (s, P(s)) \in (R \cap P).$$

By the definition of the oracle of absolute correctness:

$$\forall s \in T : \Omega(s, P(s)).$$ □

B Proof of Proposition 9

Proof *Proof of Sufficiency*. If the execution of P' for every element of T satisfies the oracle $\omega(s, s')$ then:

$$\forall s \in T : \Omega(s, P(s)) \Rightarrow \Omega(s, P'(s)).$$

Replacing $\Omega(,)$ by its definition, we find:

$$\forall s \in T : (s \in dom(R) \Rightarrow (s, P(s)) \in R) \Rightarrow (s \in dom(R) \Rightarrow (s, P'(s)) \in R).$$

The body of this quantified formula has the form: $(a \Rightarrow b) \Rightarrow (a \Rightarrow c)$. If we simplify this Boolean expression, we find that it can be written as: $(a \wedge b) \Rightarrow c$. Given that in our case b (which is $(s, P(s)) \in R$) logically implies a (which is $s \in dom(R)$)), this can further be simplified to: $(b \Rightarrow c)$. Hence, we write:

$$\forall s \in T : ((s, P(s)) \in R) \Rightarrow ((s, P'(s)) \in R).$$

Because P and P' are deterministic, this can be written as:

$$\forall s \in T : \exists s' : s' = P(s) \wedge ((s, s') \in R) \Rightarrow \exists s' : s' = P'(s) \wedge ((s, P'(s)) \in R).$$

By rewriting $s' = P(s)$ in relational form as $(s, s') \in P$ and taking the intersection, we find:

$$\forall s \in T : \exists s' : ((s, s') \in R \cap P) \Rightarrow \exists s' : ((s, P'(s)) \in R \cap P').$$

By the definition of domain, we write:

$$\forall s \in T : s \in dom(R \cap P) \Rightarrow s \in dom(R \cap P').$$

Factoring the term $(s \in T)$ into the formula, we find:

$$\forall s \in S : s \in T \wedge s \in dom(R \cap P) \Rightarrow s \in T \wedge s \in dom(R \cap P').$$

Using the same argument as the proof of the previous proposition, we find:

$$\forall s \in S : s \in dom({}_T\backslash R \cap P) \Rightarrow s \in dom({}_T\backslash R \cap P').$$

From which we infer, by rewriting in relational form:

$$({}_T\backslash R \cap P)L \subseteq ({}_T\backslash R \cap P')L.$$

In other words, P' is more-correct than P with respect to ${}_T\backslash R$.

Proof of Necessity. If P' is more-correct than P with respect to ${}_T\backslash R$ then $({}_T\backslash R \cap P)L \subseteq ({}_T\backslash R \cap P')L$, which we represent by the following logic formula:

$$\forall s \in S : s \in dom({}_T\backslash R \cap P) \Rightarrow s \in dom({}_T\backslash R \cap P').$$

If this formula holds for all s in S, it holds necessarily for all s in T.

$$\forall s \in T : s \in dom({}_T\backslash R \cap P) \Rightarrow s \in dom({}_T\backslash R \cap P').$$

By factoring out the pre-restriction from the domain, we get:

$$\forall s \in T : s \in T \wedge s \in dom(R \cap P) \Rightarrow s \in T \wedge s \in dom(R \cap P').$$

We remove the clause $(s \in T)$, which is now redundant:

$$\forall s \in T : s \in dom(R \cap P) \Rightarrow s \in dom(R \cap P').$$

Because P and P' are deterministic, this formula can be written as:

$$\forall s \in T : (s, P(s)) \in (R \cap P) \Rightarrow (s, P'(s)) \in (R \cap P').$$

Using the Boolean manipulations we showed in the previous proof, we find this to be equivalent to:

$$\forall s \in T : (s \in dom(R) \Rightarrow (s, P(s)) \in R) \Rightarrow (s \in dom(R) \Rightarrow (s, P'(s)) \in R).$$

Using the formula of the oracle of absolute correctness with respect to R, we find:
$\forall s \in T : \Omega(s, P(s)) \Rightarrow \Omega(s, P'(s))$. □

C Proof of Proposition 10

Proof *Proof of Sufficiency*. Let program P' satisfy the oracle of strict relative correctness; then according to the definition of this oracle, it satisfies the condition $(\forall s \in T : \omega(s, P'(s)))$. By proposition 9, P' is more-correct than P with respect to ${}_{T\setminus}R$, i.e., $({}_{T\setminus}R \cap P)L \subseteq_{T\setminus} R \cap P')L$. To prove strict relative correctness, we must prove that there exists an element s in the domain of $({}_{T\setminus}R \cap P')$ that is not in the domain of $({}_{T\setminus}R \cap P)$. To this effect, we consider the second clause of the oracle:

$(\exists s \in T : \neg\Omega(s, P(s)) \wedge \Omega(s, P'(s)))$.
By the definition of $\Omega(,)$, we find:
$(\exists s \in T : s \in dom(R) \wedge (s, P(s)) \notin R \wedge (s \in dom(R) \Rightarrow (s, P'(s)) \in R))$.
Using Boolean identities, we simplify this to:
$(\exists s \in T : s \in dom(R) \wedge (s, P(s)) \notin R \wedge (s, P'(s)) \in R)$.
Since $(s, P'(s)) \in R$ logically implies $s \in dom(R)$, we write:
$(\exists s \in T : (s, P(s)) \notin R \wedge (s, P'(s)) \in R)$.
From $(s \in T \wedge (s, P'(s)) \in R$ we easily infer $s \in dom({}_{T\setminus}R \cap P')$, following the same argument that we used in the proofs of propositions 8 and 9. From $(s, P(s)) \notin R$ we infer $s \notin dom(R \cap P)$, whence we infer $s \notin dom({}_{T\setminus}R \cap P)$, since ${}_{T\setminus}R \subseteq R$.

Proof of Necessity If P' is strictly more-correct than P with respect to ${}_{T\setminus}R$, then it is more-correct, hence by proposition 9, $(\forall s \in T : \omega(s, P(s)))$. On the other hand, we know that there exists an element of $dom({}_{T\setminus}R \cap P')$ that is not in $dom({}_{T\setminus}R \cap P)$. Using the same arguments cited in the proof of proposition 9, we infer: $(s, P(s)) \notin_{T\setminus} R \wedge (s, P'(s)) \in_{T\setminus} R$. From the second clause, we infer that s is in T, which we use to rewrite the formula as: $\exists s \in T : (s, P(s)) \notin R \wedge (s, P'(s)) \in R$. Using the Boolean transformation alluded to above, we find this to be equivalent to: $(\exists s \in T : \neg\Omega(s, P(s)) \wedge \Omega(s, P'(s)))$. □

D Proof of Proposition 11

We propose to prove that the following Hoare formula is valid in Hoare's deductive logic:
v: $\{(\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P)\}$

```
m=1; inc=false; Pp=P;
while (! inc && m<=M)
  {while (! smc(Pp,P) && MorePatches(P,m))
    {Pp = NextPatch(P,m);}
   if smc(Pp,P) {inc=true;}
   else {m=m+1;}}//try higher multiplicity
```

$\{Pp \sqsupset_{R'} P\}$.

Applying the sequence rule to v, with the following intermediate predicate int:

$$(\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P)$$
$$\wedge m = 1 \wedge \neg inc \wedge Pp = P$$

yields the following lemmas:

v_0: $\{(\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P)\}$

```
m=1; inc=false; Pp=P;
```

$\{(\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P) \wedge m = 1 \wedge \neg inc \wedge Pp = P\}$.

v_1: $\{(\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P) \wedge m = 1 \wedge \neg inc \wedge Pp = P\}$

```
while (! inc && m<=M)
   {while (! smc(Pp,P) && MorePatches(P,m))
     {Pp = NextPatch(P,m);}
    if smc(Pp,P) {inc=true;}
    else {m=m+1;}}//try higher multiplicity
```

$\{Pp \sqsupset_{R'} P\}$.

If we apply the (concurrent) assignment rule to v_0, we get:

$$v_{00} : (\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P) \Rightarrow (\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P) \wedge 1 = 1 \wedge \textbf{true} \wedge P = P\}$$

This formula is clearly a tautology, hence we turn our attention to v_1, to which we apply the while rule, with the following loop invariant inv:

$$inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))),$$

where $inb(m)$ (stands for: *in bounds*) is shorthand for: $1 \leq m \leq M$. Application of the while rule to v_1 with the selected loop invariant yields three lemmas:

$v_{10} : (\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P) \wedge m = 1 \wedge \neg inc \wedge Pp = P \Rightarrow inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)))$

v_{11}: $\{(\neg inc \wedge m \leq M) \wedge inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)))\}$

```
   {while (! smc(Pp,P) && MorePatches(P,m))
     {Pp = NextPatch(P,m);}
    if smc(Pp,P) {inc=true;}
    else {m=m+1;}}//try higher multiplicity
```

$\{inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)))\}$.

$$v_{12} : \neg(\neg inc \wedge m \leq M) \wedge inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))) \Rightarrow Pp \sqsupset_{R'} P$$

To check the validity of v_{10}, we rewrite it by distributing $inb(m)$ over the disjunction and replacing m by 1 on the right hand side:

$v_{10} : (\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P) \wedge m = 1 \wedge \neg inc \wedge Pp = P \Rightarrow (inb(m) \wedge inc \wedge Pp \sqsupset_{R'} P) \vee (inb(m) \wedge \neg inc \wedge (\exists h : 1 \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))$

Now it is clear that v_{10} is a tautology, since the left hand side logically implies the second disjunct of the right hand side, assuming, as we do, that $M \geq 1$. As for v_{12}, its left hand side can be simplified into $(inc \wedge Pp \sqsupset_{R'} P)$, due to the contradiction between $m > M$ and $inb(m)$, and the contradiction between inc and $\neg inc$. Hence, v_{12} is also a tautology. We turn our attention to v_{11}, which we first simplify as follows:

v_{11}: $\{\neg inc \wedge inb(m) \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)\}$

```
{while (! smc(Pp,P) && MorePatches(P,m))
  {Pp = NextPatch(P,m);}
 if smc(Pp,P) {inc=true;}
 else {m=m+1;}}//try higher multiplicity
```

$\{inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)))\}$.
We apply the sequence rule to v_{11}, with the following intermediate predicate int':

$$(Pp \sqsupset_{R'} P \vee PS(m) = \epsilon) \wedge$$
$$\neg inc \wedge inb(m) \wedge$$
$$(Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)).$$

This yields the following two lemmas:
v_{110}: $\{\neg inc \wedge inb(m) \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))\}$

```
{while (! smc(Pp,P) && MorePatches(P,m))
  {Pp = NextPatch(P,m);}
```

$\{(Pp \sqsupset_{R'} P \vee PS(m) = \epsilon) \wedge \neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)).\}$.
v_{111}: $\{(Pp \sqsupset_{R'} P \vee PS(m) = \epsilon) \wedge \neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)).\}$

```
 if smc(Pp,P) {inc=true;}
 else {m=m+1;}}//try higher multiplicity
```

$\{inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)))\}$.
We apply the while rule to v_{110}, with the following loop invariant, inv':

$$\neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)).$$

This yields the following three lemmas:
$v_{1100} : \neg inc \wedge inb(m) \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))$
$\Rightarrow$
$\neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))$.
$v_{1101} : \{\neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))$
$\wedge \neg (Pp \sqsupset_{R'} P \wedge PS(m) \neq \epsilon)\}$

```
{Pp = NextPatch(P,m);}
```

$\{\neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))\}$
$v_{1102} : \neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h$
$\leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)) \wedge (Pp \sqsupset_{R'} P \vee PS(m) = \epsilon)$
$\Rightarrow$
$(Pp \sqsupset_{R'} P \vee PS(m) = \epsilon) \wedge \neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))$.
To see that v_{1100} is a tautology, it suffices to distribute the $\wedge$ over the $\vee$ on the right hand side of the implication, and to notice that the second disjunct on the right hand side is a copy of the left hand side of the implication. As for v_{1102}, it is clearly a tautology, since the right hand side of $\Rightarrow$ is merely a copy of the left hand side. We turn our attention to v_{1101} now, and we begin by simplifying its precondition by virtue of Boolean identities:
v_{1101}: $\{\neg inc \wedge inb(m) \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P) \wedge \neg (Pp \sqsupset_{R'} P) \wedge PS(m) \neq \epsilon)\}$

```
{Pp = NextPatch(P,m);}
```

$\{\neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))\}$
We consider v_{1101}, to which we must apply the assignment statement rule; to this effect, we must analyze the semantics of function `NextPatch(P,m)`. We assume that this function performs the following operations:

```
Pp=head(PS(m)); PS(m)=tail(PS(m));
```

Hence, application of the assignment rule yields the following formula:
$v_{11010} : \neg inc \wedge inb(m)$
$\wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))$
$\wedge (\neg (Pp \sqsupset_{R'} P \wedge PS(m) \neq \epsilon)$
$\Rightarrow$
$\neg inc \wedge inb(m) \wedge (head(PS(m)) \sqsupset_{R'} P \vee$
$(\exists Q \in tail(PS(m)) : Q \sqsupset_{R'} P) \vee (\exists h : m + 1 \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))$

We consider the first two disjuncts in the parenthesized expression:
$(head(PS(m)) \sqsupset_{R'} P) \vee (\exists Q \in tail(PS(m)) : Q \sqsupset_{R'} P)$ and we merge them into a single expression:
$(\exists Q \in PS(m) : Q \sqsupset_{R'} P)$.
Now we merge this expression with the third disjunct above: $(\exists Q \in PS(m) : Q \sqsupset_{R'} P) \vee (\exists h : m+1 \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)$, to obtain: $(\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)$.
Replacing these in v_{11010}, we find that the right hand side is a logical conclusion of the left hand side, hence v_{11010} is a tautology. We now consider v_{111}, to which we apply the if-then-else rule, which yields two lemmas:
v_{1110}: $\{(Pp \sqsupset_{R'} P) \wedge (Pp \sqsupset_{R'} P \vee PS(m) = \epsilon) \wedge \neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)).\}$

```
inc=true;
```

$\{inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)))\}$.
v_{1111}: $\{\neg(Pp \sqsupset_{R'} P) \wedge (Pp \sqsupset_{R'} P \vee PS(m) = \epsilon) \wedge \neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)).\}$

```
m=m+1;
```

$\{inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)))\}$.
We simplify v_{1110} and apply the assignment rule to it, yielding:
$v_{11100} : (Pp \sqsupset_{R'} P) \wedge \neg inc \wedge inb(m) \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)$
$: \Rightarrow$
$inb(m) \wedge (Pp \sqsupset_{R'} P)$,
This is clearly a tautology. We simplify v_{1111} and apply the assignment rule to it, yielding:
$v_{11110} : \neg(Pp \sqsupset_{R'} P) \wedge PS(m) = \epsilon \wedge \neg inc \wedge inb(m) \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)$
$\Rightarrow$
$inb(m+1) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m+1 \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)))\}$.
If we know that there exists Q strictly more-correct than P in one of the patch sequences $PS(m)$, $PS(m+1)$, $...PS(M)$ but $PS(m)$ is empty, then it must be in one of the sequence $PS(m+1)$, $PS(m+2)$, $...PS(M)$. For the same reason, m is necessarily strictly less than M, since Q is somewhere in $PS(m+1)$, $PS(m+2)$, $...PS(M)$. Hence $inb(m+1)$ holds. We conclude that v_{11110} is a tautology.

Since all the lemmas generated form v are valid, so is v. Hence, `UnitIncCor()` is partially correct with respect to the specification:

- Precondition: $(\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P)$.
- Postcocndition: $Pp \sqsupset_{R'} P$.

E Proof of Proposition 12

We must prove the validity of the following formula in Hoare logic [25]:
v: {**true** }

```
m=1; inc=false; Pp=P;
while (! inc && m<=M)
  {while (! smc(Pp,P) && MorePatches(P,m))
    {Pp = NextPatch(P,m);}
   if smc(Pp,P) {inc=true;}
   else {m=m+1;}}//try higher multiplicity
```

$\{inc \Rightarrow Pp \sqsupset_{R'} P\}$.
Applying the sequence rule to v with the intermediate predicate int: $inc \Rightarrow Pp \sqsupset_{R'} P$ yields the following formulas:
v_0: {**true** }

```
m=1; inc=false; Pp=P;
```

$\{inc \Rightarrow Pp \sqsupset_{R'} P\}$.
v_1: $\{inc \Rightarrow Pp \sqsupset_{R'} P\}$

```
while (! inc && m<=M)
  {while (! smc(Pp,P) && MorePatches(P,m))
    {Pp = NextPatch(P,m);}
   if smc(Pp,P) {inc=true;}
   else {m=m+1;}}//try higher multiplicity
```

$\{inc \Rightarrow Pp \sqsupset_{R'} P\}$.
The (concurrent) assignment rule applied to v_0 yields:
v_{00}: **true** $\Rightarrow$ (**false** $\Rightarrow P \sqsupset_{R'} P$),
which is a tautology. We apply the while rule to v_1 with the loop invariant inv: $inc \Rightarrow Pp \sqsupset_{R'} P$, which yields the following formulas:
v_{10}: $(inc \Rightarrow Pp \sqsupset_{R'} P) \Rightarrow (inc \Rightarrow Pp \sqsupset_{R'} P)$
v_{11}: $\{(inc \Rightarrow Pp \sqsupset_{R'} P) \wedge (\neg inc \wedge m \leq M)\}$

```
{while (! smc(Pp,P) && MorePatches(P,m))
  {Pp = NextPatch(P,m);}
 if smc(Pp,P) {inc=true;}
 else {m=m+1;}}//try higher multiplicity
```

$\{inc \Rightarrow Pp \sqsupset_{R'} P\}$.
v_{12}: $(inc \Rightarrow Pp \sqsupset_{R'} P) \wedge (inc \vee m > M) \Rightarrow (inc \Rightarrow Pp \sqsupset_{R'} P)$.
Formulas v_{10} and v_{12} are clearly tautologies; we apply the sequence rule to v_{11}, with int: $inc \Rightarrow Pp \sqsupset_{R'} P$, which yields the following formulas:
v_{110}: $\{(inc \Rightarrow Pp \sqsupset_{R'} P) \wedge (\neg inc \wedge m \leq M)\}$

```
while (! smc(Pp,P) && MorePatches(P,m))
  {Pp = NextPatch(P,m);}
```

$\{inc \Rightarrow Pp \sqsupset_{R'} P\}$
v_{111}: $\{(inc \Rightarrow Pp \sqsupset_{R'} P)\}$

```
 if smc(Pp,P) {inc=true;}
 else {m=m+1;}//try higher multiplicity
```

$\{inc \Rightarrow Pp \sqsupset_{R'} P\}$.
We apply the while rule to v_{110} with the loop invariant inv: $\neg inc$, which yields the following formulas:
v_{1100}: $(inc \Rightarrow Pp \sqsupset_{R'} P) \wedge (\neg inc \wedge m \leq M) \Rightarrow \neg inc$.
v_{1101}: $\{\neg inc \wedge (\neg Pp \sqsupset_{R'} P \wedge MorePatches(P, m))\}$

```
  {Pp = NextPatch(P,m);}
```

$\{\neg inc\}$.
v_{1102}: $\neg inc \wedge \neg(\neg Pp \sqsupset_{R'} P \wedge MorePatches(P, m)) \Rightarrow (inc \Rightarrow Pp \sqsupset_{R'} P)$.
Formula v_{1100} is clearly a tautology; formula v_{1102} is also a tautology because it has the form $((\neg a \wedge b) \Rightarrow (a \Rightarrow c))$, which can be simplified as $(a \vee \neg b) \vee (\neg a \vee c)$; we focus on v_{1101}, to which we apply the assignment statement rule, which yields:
v_{11010}: $(\neg inc \wedge (\neg Pp \sqsupset_{R'} P \wedge MorePatches(P, m))) \Rightarrow \neg inc$.
This is clearly a tautology; we turn our attention to v_{111}, to which we apply the if-then-else rule, which yields:
v_{1110}: $\{(inc \Rightarrow Pp \sqsupset_{R'} P) \wedge (Pp \sqsupset_{R'} P)\}$

```
 {inc=true;}
```

$\{inc \Rightarrow Pp \sqsupset_{R'} P\}$.
v_{1111}: $\{(inc \Rightarrow Pp \sqsupset_{R'} P) \wedge \neg(Pp \sqsupset_{R'} P)\}$

```
 {m=m+1;}
```

$\{inc \Rightarrow Pp \sqsupset_{R'} P\}$.
Application of the asignment statement rule to v_{1110} and v_{1111} yields, respectively:
v_{11100}: $(inc \Rightarrow Pp \sqsupset_{R'} P) \wedge (Pp \sqsupset_{R'} P) \Rightarrow (Pp \sqsupset_{R'} P)$.
v_{11110}: $(inc \Rightarrow Pp \sqsupset_{R'} P) \wedge \neg(Pp \sqsupset_{R'} P) \Rightarrow (inc \Rightarrow Pp \sqsupset_{R'} P)$.
Formulas v_{11100} and v_{11110} are both tautologies. This concludes the proof that
v: {**true** }

```
UnitIncCor()
```

$\{inc \Rightarrow Pp \sqsupset_{R'} P\}$
is valid in Hoare's logic. Hence, `UnitIncCor()` is partially correct with respect to the specification defined by the following pre/post condition pair:

- Precondition: **true** .
- Postcondition: $inc \Rightarrow Pp \sqsupset_{R'} P$.

References

1. Abreu, R.: Gzoltar: a toolset for automatic test suite minimization and fault identification. In: International Workshop on the Future of Debugging, Lugano, Switzerland (2013)
2. Abrial, J.R.: The B Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
3. Anonymous: Addendum, the bane of generate-and-validate program repair, crcfix data. Technical report. https://anonymous.4open.science/r/7c54e6e6-1c2f-491c-bf5a-d7f451fb463c/ (May 2020)
4. Anonymous: Addendum, the bane of generate-and-validate program repair, crcfix tool. Technical report. https://anonymous.4open.science/r/95d330a9-97bf-44ab-9144-f214dce174d2/ (September 2020)
5. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. **1**(1), 11–33 (2004)
6. Bergstra, J.A.: Instruction sequence faults with formal change justification. Sci. Ann. Comput. Sci. **30**(2), 105–166 (2020)
7. Boudriga, N., Elloumi, F., Mili, A.: The lattice of specifications: applications to a specification methodology. Formal Aspects Comput. **4**(6), 544–571 (1992)
8. Brink, C., Kahl, W., Schmidt, G.: Relational Methods in Computer Science. Advances in Computer Science. Springer, Berlin (1997)
9. Christakis, M., Heizmann, M., Mansur, M.N., Schilling, C., Wuestholz, V.: Semantic fault localization and suspiciousness ranking. In: Vojnar, T., Zhang, L. (eds.) Proceedings, TACAS 2019, Number 11427 in LNCS, pp. 226–243 (2019)
10. Debroy, V., Eric Wong, W.: Combining mutation and fault localization for automated program debugging. J. Syst. Softw. **90**, 45–60 (2013)
11. DeMarco, F., Xuan, J., Berra, D.L., Monperrus, M.: Automatic repair of buggy if conditions and missing preconditions with SMT. In: Proceedings, CSTVA, pp. 30–39 (2014)
12. Demarco, F., Xuan, J., Berre, D.L., Monperrus, M.: Automatic repair of buggy if conditions and missing preconditions with SMT. In Proceedings, CSTVA, pp. 30–39 (2014)
13. Desharnais, J., Diallo, N., Ghardallou, W., Frias, M.F., Jaoua, A., Mili, A.: Relational mathematics for relative correctness. In: RAMICS, 2015, volume 9348 of LNCS, Braga, Portugal. Springer, pp 191–208, September (2015)
14. Desharnais, J., Diallo, N., Ghardallou, W., Ali, M.: Definitions and implications. In: Science of Computer Programming, Projecting programs on specifications (2017)
15. Diallo, N., Ghardallou, W., Desharnais, J., Frias, M., Jaoua, A., Mili, A.: What is a fault? and why does it matter? ISSE **19**, 219–239 (2017)
16. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Englewood Cliffs (1976)
17. Ermis, E., Schaef, M., Wies, T.: Error invariants. In: Giannakopoulou, D., Mery, D. (eds.) Proceedings, FM 2012, Number 7436 in LNCS, pp. 187–201 (2012)
18. Frenkel, H., Grumberg, O., Pasareanu, C., Sheinvald, S.: Assume, guarantee or repair. In: Biere, A., Parker, D. (eds.) Proceedings, TACAS 2020, Number 12078 in LNCS. Springer (2020)
19. Gazzola, L., Micucci, D., Mariani, L.: Automatic software repair: a survey. IEEE Trans. Soft. Eng. **45**(1), 34–67 (2019)
20. Ghardallou, W., Diallo, N., Mili, A., Frias, M.: Debugging without testing. In: Proceedings, International Conference on Software Testing, Chicago, IL (April 2016)
21. Gopinath, R., Alipour, A., Ahmed, I., Jensen, C., Groce, A.: Measuring effectiveness of mutant sets. In: Proceedings, Ninth International Conference on Software Testing, Chicago, IL, April 11–15 (2016)
22. Gries, D.: The Science of Programming. Springer, New York (1981)
23. Gupta, R., Pal, S., Kanade, A., Shevade, S.K.: Deepfix: Fixing common c language errors by deep learning. In: Proceedings, AAAI, pp. 1345–1351 (2017)
24. Hehner, E.C.R.: A Practical Theory of Programming. Prentice Hall, Englewood Cliffs (1992)
25. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–583 (1969)
26. Hoare, C.A.R.: Unified theories of programming. In: Mathematical Methods in Program Development. Springer (1997)
27. IEEE Std 7-4.3.2-2003. Ieee standard criteria for digital computers in safety systems of nuclear power generating stations. Technical report, The Institute of Electrical and Electronics Engineers (2003)
28. Jiang, J.J., Xiong, Y.F., Zhang, H.Y., Gao, Q., Chen, X.C.: Shaping program repair space with existing patches and similar code. In: Proceedings, ISSTA, pp. 298–309 (2018)
29. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Proceedings, PLDI, pp. 437–446 (2011)
30. Just, R., Jalali, D., Ernst, M.D.: Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings. ISSTA 2014, pp. 437–440. CA, USA, San Jose (July 2014)

31. Ke, Y., Stolee, K.T., Le Goues, C., Brun, Y.: Repairing programs with semantic code search. In: International Conference on Automated Software Engineering (2015)
32. Khaireddine, B., Martinez, M., Mili, A.: Program repair at arbitrary fault depth. In: Proceedings, ICST 2019, Xi'an, China (April 2019)
33. Khaireddine, B., Mili, A.: Quantifying faultiness: What does it mean to have *n* faults? In: Proceedings, FormaliSE 2021, ICSE 2021 Colocated Conference (May 2021)
34. Khaireddine, B., Zakharchenko, A., Mili, A.: A generic algorithm for program repair. In: Proceedings, FormaliSE, Buenos Aires, Argentina (May 2017)
35. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: International Conference on Software Engineering (ICSE), pp. 802–811 (2013)
36. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: ICSE, pp. 802–811 (2013)
37. Koyuncu, A., Liu, K., Bissiande, T.F., Kim, D., Klein, J., Monperrus, M., LeTraon, Y.: Fixminer: Mining relevant fix patterns for automated program repairs. In: Empirical Software Engineering, pp. 1–45 (2020)
38. Laprie, J.C.: Dependable computing: concepts, challenges, directions. In: Proceedings, COMPSAC (2004)
39. Le, X.-B.D., Chu, D.-H., Lo, D., Goues, C.L., Visser, W.: S3: Syntax and semantic guided repair synthesis via programming examples. In Proceedings, FSE 2017, Paderborn, Germany, September 4–8 (2017)
40. LeGoues, C., Forrest, S., Weimer, W.: Current challenges in automatic software repair. Softw. Qual. J. **21**(3), 421–443 (2013)
41. LeGoues, C., Dewey, V.M., Forrest, S., Weimer, W.: A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In: Proceedings, ICSE 2012, pp. 3–13 (2012)
42. Li, Y., Wang, S., Nguyen, T.N.: Dlfix: context-based code transformation learning for automated program repair. In: Proceedings, ICSE 2020, Seoul, South Korea (May 2020)
43. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: a generic method for automated software repair. IEEE Trans. Softw. Eng. **31**(1) (2012)
44. Lin, D., Koppel, J., Chen, A., Solar-Lezma, A.: Quixbugs: a multilingual program repair benchmark set based on the quixey challenge. In: Proceedings, SPALSH (2017)
45. Le Goues, C., Pradel, M., Roychoudhury, A.: Automated program repair. Commun. ACM **62**(12), 56–65 (2019)
46. Liu, K., et al.: Lsrepair: Live search of fix ingredients for automated program repair. In: Proceedings, 25th Asia-Pacific Software Engineering Conference. IEEE (2018)
47. Long, F., Rinard, M.: Prophet: automatic patch generation via learning from successful patches. Technical Report Technical Report MIT-CSAIL-TR-2015, MIT (2015)
48. Long, F., Rinard, M.: Staged program repair with condition synthesis. In: Proceedings, ESEC-FSE, (2015)
49. Long, F., Rinard, M.: Staged program repair with condition synthesis. In: ESEC-FSE, (2015)
50. Long, F., Rinard, M.: An analysis of the search spaces for generate-and-validate patch generation systems. In: ICSE 2016 (2016)
51. Lou, Y., Ghanbari, A., Li, X., Zhang, L., Zhang, H., Hao, D., Zhang, L.: Can automated program repair refine fault localization? A unified debugging approach. In: Proceedings, ISSTA, pp. 75–87 (2020)
52. Manna, Z.: A Mathematical Theory of Computation. McGraw-Hill, New York (1974)
53. Martinez M., Monperrus M.: Mining software repair models for reasoning on the search space of automated program fixing. In: Empirical Software Engineering (2013)
54. Martinez, M., Monperrus, M.: Astor: a program repair library for java. In: Proceedings. ISSTA 2016, pp. 441–444. Saarbrucken, Germany (2016)
55. Martinez, M., Monperrus, M.: Astor: exploring the design space of generate-and-validate program repair beyond genprog (2018)
56. Martinez, M., Monperrus, M.: Ultra large repair search space with automatically mined templates: the cardumen mode of astor. In: Proceedings, SSBSE, pp. 65–86 (2018)
57. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Proceedings, ICSE 2016, Austin, TX (May 2016)
58. Mili, A., Frias, M., Jaoua, A.: On faults and faulty programs. In: Hoefner, P., Jipsen, P., Kahl, W., Mueller, M.E. (eds.) Proceedings, RAMICS 2014, Volume 8428 of LNCS, pp. 191–207 (2014)
59. Mills, H.D., Basili, V.R., Gannon, J.D., Hamlet, D.R.: Structured Programming: A Mathematical Approach. Allyn and Bacon, Boston (1986)
60. Monperrus, M.: A critical review of patch generation learned from human written patches: essay on the problem statement and evaluation of automatic software repair. In: Proceedings, ICSE 2014, Hyderabad, India (2014)
61. Morgan, C.C.: Programming from Specifications, International Series in Computer Sciences, 2nd edn. Prentice Hall, London (1998)
62. Musa, J.D.: Operational profile in software reliability engineering. IEEE Softw. **10**(2), 14–32 (1993)

63. Nguyen, H.D.T., Qi, D.W., Roychoudhury, A., Chandra, S.: Semfix: Program repair via semantic analysis. In: Proceedings, ICSE, pp. 772–781 (2013)
64. Nilizadeh, A., Calvo, M., Leavens, G.T., Cok, D.R.: Generating counter examples in the form of unit tests from hoare-style verification attempts. In: Proceedings, 32nd IEEE/ACM International Conference on Formal Methods in Software Engineering, pp. 124–128. IEEE/ACM, Pittsburgh, PA (2022)
65. Nilizadeh, A., Calvo, M., Leavens, G.T., Le, X.-B.D.: More reliable test suites for dynamic APR by using counter-examples. In: Proceedings, 32nd IEEE International Symposium on Software Reliability Engineering, pp. 208–219. IEEE (2021)
66. Nilizadeh, A., Leavens, G.T., Le, X.-B.D., Pasareanu, C.S., Cok, D.R.: Exploring true test overfitting in dynamic automated program repair using formal methods. In: Proceedings, 14th IEEE International Conference on Software Testing, Verification and Validation, pp. 229–240. IEEE (2021)
67. Qi, Z., Long, F., Achour, S., Rinard, M.: An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Proceedings, ISSTA 2015, Baltimore, MD, July (2015)
68. Rothenberg, B.-C., Grumberg, O.: Sound and complete mutation-based program repair. In: Proceedings, FM, pp. 593–611 (2016)
69. Rothenberg, B.-C., Grumberg, O.: Must fault localization for program repair. In: Proceedings, CAV, pp. 658–680 (2020)
70. Saha, S., Saha, R., Prasad, M.: Harnessing evolution for multi-hunk program repair. In: Proceedings, ICSE (2019)
71. Soto, M., Le Goues, C.: Using a probabilistic model to predict bug fixes. In: Proceedings, SANER, pp. 221–231 (2018)
72. Tan, S.H., Roychoudhury, A.: Relifix: Automated repair of software regressions. In: ICSE (2015)
73. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings, International Conference on Software Engineering (ICSE), pp. 364–374 (2009)
74. Wen, W., Chen, J.J., Wu, R., Hao, D., Cheung, S.C.: Context-aware patch generation for better automated program repair. In: Proceedings, ICSE 2018, Gothenburg, Sweden, May 27–June 3 (2018)
75. Wong, W.R., Gao, R., Li, Y.H., Abreu, R., Wotawa, F.: A survey of software fault localization. IEEE Trans. Softw. Eng. **42**, 707–740 (2016)
76. Xin, Q., Reiss, S.P.: Leveraging syntax-related code for automated program repair. In: Proceedings, ASE 2017, Urbana Champaign, IL, October 30–November 3 (2017)
77. Xiong, Y.F., Wang, J., Yan, R.F., Zhang, J.C., Han, S., Huang, G., Zhang, L.: Precise condition synthesis for program repair. In Proceedings, ICSE, pp. 416–426 (2017)
78. Xuan, J., Martinez, M., Demarco, F., Clement, M., Lamelas Marcotte, S., Durieux, T., LeBerre, D., Monperrus, M.: Nopol: Automatic repair of conditional statement bugs in java programs. In: IEEE-TSE (2016)
79. Xuan, J., Monperrus, M.: Test case purification for improving fault localization. In: Proceedings, FSE (2014)